



Issue 1
October 1984

AT&T 3B2 Computer UNIX™ System V Release 2.0 Terminal Information Utilities Guide

Select Code
305-424

Comcode
403778392

Copyright © 1984 AT&T Technologies, Inc.
All Rights Reserved
Printed in U.S.A.

TRADEMARKS

The following is a listing of the trademarks that are used in this manual:

- TELETYPE — Registered trademark of Teletype Corporation
- UNIX — Trademark of AT&T Bell Laboratories

NOTICE

The information in this document is subject to change without notice. AT&T Technologies assumes no responsibility for any errors that may appear in this document.

AT&T 3B2 COMPUTER
TERMINAL INFORMATION
UTILITIES GUIDE

Update Issue 1
December 10, 1984

This update inventory should be placed behind the *3B2 Computer Terminal Information Utilities Guide* title page. This inventory identifies the pages that have been added or changed by the *3B2 Computer Terminal Information Utilities Guide Update* (305-403), dated December 10, 1984.

Revised Page(s)	Date
curses(3X) manual pages	11/84
terminfo(4) manual pages	11/84
tput(1) manual page	11/84

NOTE

This Utilities Guide contains descriptive information and UNIX* System manual pages for the commands included in one of the utilities provided with your 3B2 Computer. Since this utilities is provided with the 3B2 Computer, the manual pages have already been filed in the *3B2 Computer UNIX System V User Reference Manual*. If you do not need duplicate copies of these manual pages, they may be discarded.

A UTILITIES binder is provided with the 3B2 Computer for you to keep the descriptive information from all the Utilities Guides together. Remove the descriptive information from the soft cover, place the provided tab separator in front of the title page, and file this material in the UTILITIES binder. As previously mentioned, UNIX System manual pages may be destroyed.

If you ordered extra copies of this Utilities Guide, they should be left in the individual soft covers.

*. Trademark of AT&T Bell Laboratories

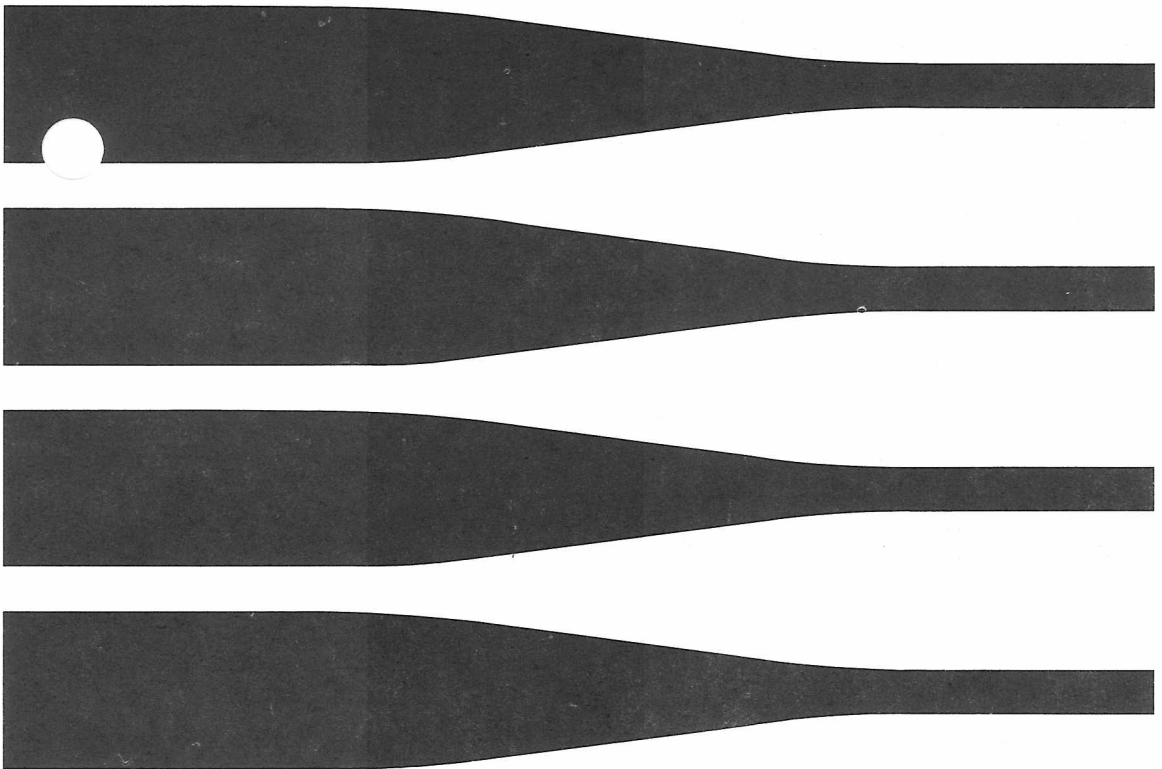


Issue 1
October 1984

**AT&T 3B2 Computer
UNIX™ System V Release 2.0
Terminal Information
Utilities Software
Information Bulletin**

Select Code
305-361

Comcode
403778210



Copyright © 1984 AT&T Technologies, Inc.
All Rights Reserved
Printed in U.S.A.

NOTE

This Software Information Bulletin (SIB) should be filed in the *3B2 Computer Owner/Operator Manual*. A tab separator, labeled "SOFTWARE INFORMATION BULLETINS," has been placed at the back of the *Owner/Operator Manual* in order to provide a convenient place for filing SIB's. Place the tab separator provided with this SIB in front of the title page and file this material behind the SOFTWARE INFORMATION BULLETINS tab separator in the *Owner/Operator Manual*.

TERMINAL INFORMATION SOFTWARE INFORMATION BULLETIN

INTRODUCTION

This Software Information Bulletin provides important information concerning the Terminal Information Utilities. Please read this bulletin carefully before attempting to install or use these utilities.

The AT&T 3B2 Computer Terminal Information Utilities are for use by an experienced C Programming Language user who needs to write screen-oriented programs. The Terminal Information Utilities are part of the UNIX* System V Release 2.0 configuration provided with all 3B2 Computers.

FEATURE DESCRIPTION

The Terminal Information Utilities provide the **terminfo** database and the **curses** screen manipulation capabilities. The **terminfo** database is used by **curses** and other UNIX System processes (the *vi* editor for example) to identify your terminal to the computer. This identification allows the computer to send the correct character sequences so that the terminal acts the way the UNIX System processes were written for it to act.

SOFTWARE DEPENDENCIES

The Terminal Information Utilities are independent of all optional utilities.

* Trademark of AT&T Bell Laboratories

NOTES ON USING UTILITIES

Curses Programs

The **curses** programs must be written and compiled like C Language programs. The command line for compiling a **curses** program is as follows:

```
cc program.c -lcurses -o program
```

mvscanw() Function

The **mvscanw()** function will not compile. It will generate the following message:

```
undefined symbol _sscans
```

savetty Function

The **savetty** function will return the value -2138557068 instead of the correct value.

DOCUMENTATION

This Software Information Bulletin should be inserted in the *3B2 Computer Owner/Operator Manual*.

The commands provided by the Terminal Information Utilities are described in the *3B2 Computer Terminal Information Utilities Guide*. Also provided in this guide are various examples of **curses** programs and **terminfo** capabilities.

RELEASE FORMAT

Storage Structure

The Terminal Information Utilities (commands) are installed in the **/etc** and **/usr** directories.

System Requirements

The minimum equipment configuration required for the use of the Terminal Information Utilities is 0.5 megabytes of random access memory and a 10-megabyte hard disk.

To install the Terminal Information Utilities software there must be 150 free blocks of storage in **root** file system and between 556 and 1175 free blocks of storage in the **/usr** file system. The exact value of storage needed in **/usr** depends on the number of terminal types installed for the **terminfo** database. Adequate storage space is checked automatically as part of the installation process. The installation process installs the utilities only if adequate storage space is available.

Files Delivered

The Terminal Information Utilities for the 3B2 Computer are delivered on two floppy disks. One floppy disk contains the list of supported terminals for the **terminfo** database. This floppy disk also contains a menu so that you may select the desired terminals for your database. The other floppy disk contains the commands and files needed by the utilities. The directory structure and files for this floppy disk are as follows.

DIRECTORY	FILES
/etc	termcap
/usr/bin	tic tput
/usr/include	curses.h term.h unctrl.h
/usr/lib	libcurses.a

UTILITIES INSTALL PROCEDURE

Use the standard software install procedure described in the *3B2 Computer Owner/Operator Manual* for the installation of the Terminal Information Utilities.

UTILITIES REMOVE PROCEDURE

Use the standard software remove procedure described in the *3B2 Computer Owner/Operator Manual* for the removal of the Terminal Information Utilities.

CONTENTS

Chapter 1.	INTRODUCTION
Chapter 2.	SCREEN MANIPULATION
Chapter 3.	WINDOW MANIPULATION
Chapter 4.	MULTIPLE TERMINALS
Chapter 5.	PORTABILITY FUNCTIONS
Chapter 6.	LOWER LEVEL FUNCTIONS
Chapter 7.	TERMINFO DATABASE
Appendix A.	MANUAL PAGES
Appendix B.	CURSES EXAMPLES

Chapter 1

INTRODUCTION

	PAGE
FEATURE DESCRIPTION	1-1
GUIDE ORGANIZATION	1-2
SPECIAL NOTATIONS	1-3
Functions	1-3
Comments	1-4

Chapter 1

INTRODUCTION

This guide describes the Terminal Information Utilities available with the AT&T 3B2 Computer. These utilities use the **terminfo** database and the **curses** library to optimize the output of data to a video display terminal.

FEATURE DESCRIPTION

The **terminfo** database contains the descriptions of over 150 terminals. The terminals are described by giving a set of terminal capabilities and by describing how operations are performed. It is based on the **termcap** database, but contains a number of improvements and extensions.

The **curses** library is a collection of routines with the major function of "cursor optimization." **Curses** uses the **terminfo** database to obtain the specific information about the characteristics of any terminal that may be using the optimization function.

The capabilities described in this guide are intended for the sophisticated user with C Language programming experience who must write a screen-oriented program using the **curses** routines.

The **curses** programs are compiled as C Language programs. The general command line to compile a **curses** program is as follows:

```
cc filename.c -lcurses -o FILENAME
```

The *filename.c* variable is the name of the C Language program. The executable output of the compiled program is written to *FILENAME*.

GUIDE ORGANIZATION

This guide is structured so you may easily find the information that you are looking for without having to read the entire text. The remainder of this guide is organized as follows:

- Chapter 2, "SCREEN MANIPULATION," describes the screen manipulation capabilities provided by this utilities using examples of actual routines.
- Chapter 3, "WINDOW MANIPULATION," describes the window manipulation functions of the Terminal Information Utilities. Included in this chapter is a description of the pad manipulation capabilities.
- Chapter 4, "MULTIPLE TERMINALS," describes the procedures for accessing and using multiple terminals with the **curses** library functions.
- Chapter 5, "PORTABILITY FUNCTIONS," provides a description of the portability functions associated with **curses**.
- Chapter 6, "LOWER LEVEL FUNCTIONS," gives a brief explanation of those functions which do not need the screen optimization capabilities of **curses**. These functions are considered to be "lower level functions."

- Chapter 7, "TERMINFO DATABASE," describes the format and construction process of a **terminfo** entry.
- Appendix A, "MANUAL PAGES," contains the Terminal Information Utilities UNIX* System manual pages.
- Appendix B, "CURSES EXAMPLES," contains some larger, more useful examples of **curses** programs.

SPECIAL NOTATIONS

The following special notations and naming conventions are used throughout this guide.

Functions

The function names are shown in bold type followed by parentheses. The parentheses are used to designate the parameters to the function. The variables found in the parentheses represent the information which dictates where or how the function acts. The common variables used inside the parentheses and a description of what they represent are as follows:

- | | |
|------------|--|
| bf | Represents a Boolean flag with a value of <i>TRUE</i> or <i>FALSE</i> to indicate whether to enable or disable the function. Most functions which use this variable are initially <i>FALSE</i> . |
| ch | Represents a character entry. |
| win | Represents the window designation for the function. When this variable appears, you must specify which window is to receive the action. |

* Trademark of AT&T Bell Laboratories

INTRODUCTION

y, x Represent the row and column, respectively, for the cursor position.

str Represents a string of characters to be input.

Any other function variable that is encountered will be explained in the associated discussion.

Also, many function names include the main function with a prefix. The prefix is used to indicate the form of the function used. For instance, **printw** is a standard **curses** print routine. The **wprintw** function is a window print routine. A prefix of *mv* indicates a starting position to be given to the routine. Thus, **mvprintw** will begin printing at the designated cursor position and **mvwprintw** will begin printing at the designated cursor position inside the designated window.

Comments

Example programs contained in this guide include comments to help explain the procedures in the program. The comments are contained by the characters `/*` and `*/`. Comments are ignored by program executions.

Chapter 2

SCREEN MANIPULATION

	PAGE
STRUCTURE	2-1
INITIALIZATION	2-3
General	2-3
INPUT/OUTPUT FUNCTIONS	2-4
General	2-4
Output	2-6
Input	2-14
Delays	2-16
TERMINAL SETTINGS	2-18
Terminal Mode Setting	2-18
Option Setting	2-21

Chapter 2

SCREEN MANIPULATION

Through **curses**, a terminal screen can be scrolled, cleared, or divided into separate windows. The most basic function of a **curses** program is screen manipulation. The structure of a **curses** program is basically always the same. The simplicity or complexity of the program function has no influence on the program structure.

STRUCTURE

All programs using **curses** should include the file *curses.h*. This file defines several **curses** functions as macros and defines several global variables and the datatype *WINDOW*. References to windows are always of the type *WINDOW *name-of-window*. Curses also defines *WINDOW ** constants *stdscr* and *curscr*. The *stdscr* (standard screen) constant is used as a default value to routines expecting a window. The *curscr* (current screen) constant is used only for certain low-level operations like clearing and redrawing a garbaged screen. Integer constants, which dictate the size of the screen, for *LINES* and *COLS* are defined. Constants *TRUE* and *FALSE* are defined with values 1 and 0, respectively. Additional constants which are values returned from most **curses** functions are *ERR* and

OK. The *OK* value is returned if the function could be properly completed, and *ERR* is returned if there was some error such as moving the cursor outside of a window.

The include file *curses.h* automatically includes *stdio.h* and an appropriate tty driver interface file, currently either *termio.h* or *sgtty.h*.

Note: The driver interface *sgtty.h* is a tty driver interface used in other versions of the UNIX System.

Including *stdio.h* again is harmless but wasteful; including *termio.h* again will usually result in a fatal error.

A program using **curses** should include the loader option *-lcurses* in the makefile. This is true for both the **terminfo** level and the **curses** level.

INITIALIZATION

General

The following functions are called when initializing a program.

initscr()

The first function called should always be **initscr**. This will determine the terminal type and initialize **curses** data structures. The **initscr** function also arranges that the first call to **refresh** will clear the screen.

endwin()

A program should always call **endwin** before exiting. This function will restore tty modes, move the cursor to the lower left corner of the screen, reset the terminal into the proper nonvisual mode, and tear down all appropriate data structures.

The following is a sample of the smallest possible **curses** program. The only thing this program will do is clear (refresh) the screen.

```
#include    <curses.h>

main( )
{
    initscr();
    refresh();
    endwin();
}
```

INPUT/OUTPUT FUNCTIONS

General

printw(fmt, args)

wprintw(win, fmt, args)

mvprintw(y, x, fmt, args)

mvwprintw(win, y, x, fmt, args)

These functions correspond to **printf**. The characters which would be output by **printf** are instead output using **waddch** on the given window. Never use **printf** in a **curses** program because **curses** must have total control of the terminal.

refresh()

wrefresh(win)

These functions must be called to get any output on the terminal, since other routines merely manipulate data structures. The **wrefresh** function copies the named window to the physical terminal screen, taking into account what is already there in order to do optimization. The **refresh** function is the same, using **stdscr** as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the cursor for that window. (Refer to the "Option Setting" heading in the "TERMINAL SETTINGS" section of this chapter for a description of **leaveok**.)

move(y, x)

wmove(win, y, x)

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until **refresh** is called. The position specified is relative to the upper left corner of the window. Thus, if you have a window which is not in the upper left corner of the screen, you would have to specify the screen coordinates as a distance from the upper left corner of the window. The upper left corner of the window is position **(0, 0)**.

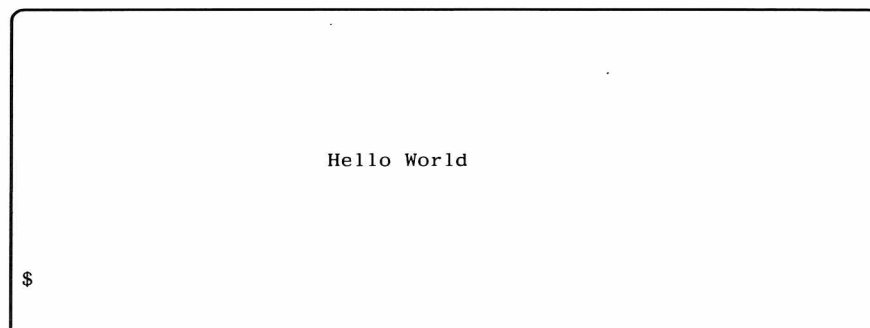
The following sample program shows the use of the **printw** and **move** functions. This program will print "Hello World" in the center

of the screen. The center of the screen is determined as the half-screen point both vertically ($\text{LINES} / 2$) and horizontally ($\text{COLS} / 2$).

```
#include    < curses.h >

main( )
{
    initscr( );
    move( LINES / 2, COLS / 2 );
    printw( "Hello World" );
    refresh( );
    endwin( );
}
```

The output of this program will appear as follows:



```

      Hello World
$
```

erase()

werase(win)

These functions copy blanks to every position in the window.

clear()

wclear(win)

These functions are like **erase** and **werase**, but they also call **clearok**, arranging that the screen will be cleared on the next call to **refresh** for that window.

clrtoobot()

wclrtoobot(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

clrtoeol()

wclrtoeol(win)

The current line to the right of the cursor is erased.

The following sample program shows the use of the **clrtoobot** function. This program will print a full screen of lines, move the cursor to the fifth line (cursor position 5, 0), and clear all lines below the cursor.

```
#include    <curses.h>

main( )
{
    int    i;

    initscr( );
    for( i=0; i<LINES; i++ )
        printw( "This is line %d of the standard screen.\n", i );
    refresh( );
    sleep(5);
    move( 5, 0 );
    clrtoobot( );
    refresh( );
    endwin( );
}
```

Output

A program using **curses** always starts by calling **initscr()**. Other modes can then be set as needed by the program. Possible modes include **cbreak()** and **idlok(stdscr, TRUE)**. These modes will be explained later in this section. During the execution of the program, output to the screen is done with routines such as **addch(ch)** and **printw(fmt,args)**. (These routines behave just like **putchar** and **printf** except that they go through **curses**.) The cursor can be moved with the call **move(row,col)**. These routines only output to a

data structure called a *window*, not to the actual screen. A window is a representation of a terminal screen containing such things as an array of characters to be displayed on the screen, a cursor, a current set of video attributes, and various modes and options. You don't need to worry about windows unless you use more than one of them, except to realize that a window is buffering your requests to output to the screen.

In order to determine how to update the screen, **curses** must know what is on the screen at all times. This requires **curses** to clear the screen in the first call to **refresh** and to know the cursor position and screen contents at all times. To send all accumulated output, it is necessary to call **refresh()**. (This can be thought of as a **flush**.) Remember, before the program exits, it should call **endwin()**, which restores all terminal settings and positions the cursor at the bottom of the screen.

See the program **scatter** in Appendix B for an example program which uses many of the output routines.

No output to the terminal actually happens until **refresh** is called. Instead, routines such as **move** and **addch** draw on a window data structure called **stdscr** (standard screen). **Curses** always keeps track of what is on the physical screen as well as what is in **stdscr**.

When **refresh** is called, **curses** compares the two screen images and sends a stream of characters to the terminal that will turn the current screen into what is desired. **Curses** considers many different ways to do this, taking into account the various capabilities of the terminal and the similarities between what is on the screen and what is desired. **Curses** usually outputs as few characters as is possible. This function is called *cursor optimization*, and it is the source of the name of the **curses** utilities.

Note: Due to the hardware scrolling of some terminals, writing to the lower right-hand character position could be impossible.

Bells and Flashing Lights

utilities()

flash()

These functions are used to signal the programmer. The **beep** function will sound the audible alarm on the terminal, if possible; and if not possible, the **beep** function will flash the screen (visible bell), if that is possible. The **flash** function will flash the screen; and if that is not possible, the **flash** function will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen. In the program **scatter** note the call to **flash()**, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement, and flashing is particularly useful if the bell bothers someone within hearing distance of the user's terminal. The routine **beep()** can be called when a real beep is desired. (If for some reason the terminal is unable to beep but able to flash, a call to **beep** will flash the screen.)

Inserting and Deleting Text

insertln()

wininsertln(win)

A blank line is inserted above the current line. The bottom line is lost. This does not necessarily imply use of the hardware insert line feature.

deleteln()

wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not necessarily imply use of the hardware delete line feature.

Writing One Character

addch(ch)
waddch(win, ch)
mvaddch(y, x, ch)
mvwaddch(win, y, x, ch)

The character **ch** is put in the window at the current cursor position of the window. If **ch** is a tab, newline, or backspace the cursor will be moved appropriately in the window. If **ch** is a different control character, it will be drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region will be scrolled up one line, and **wrefresh** will be called. If **scrollok** isn't enabled, the cursor will print subsequent characters on the last line of the window. (Refer to the "Option Setting" heading in the "TERMINAL SETTINGS" section of this chapter for the description of **scrollok**.)

The **ch** parameter is actually an integer, not a character. Video attributes can be combined with a character by OR-ing them into the parameter. This will result in these video attributes also being set, in addition to any current attributes in the window. (The intent here is that text including attributes can be copied from one place to another with **inch** and **addch**.)

Writing a String

addstr(str)
waddstr(win, str)
mvaddstr(y, x, str)
mvwaddstr(win, y, x, str)

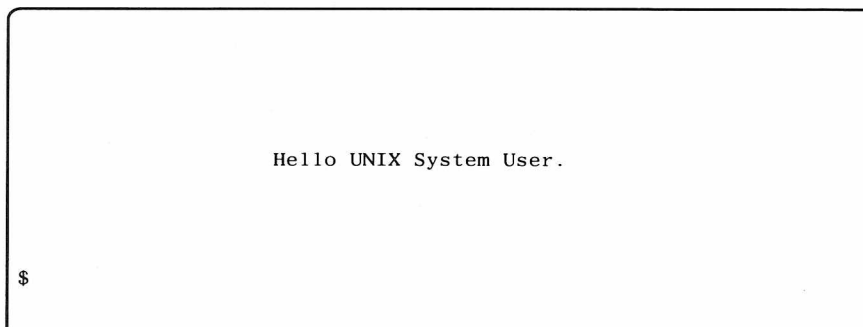
These functions write all the characters of the null terminated character string **str** on the given window. They are identical to a series of calls to **addch**.

The following sample shows the use of the **mvaddstr** function.

```
#include    <urses.h>

main( )
{
    initscr();
    mvaddstr( LINES / 2, ( COLS / 2 ) - 11,
              "Hello UNIX System User." );
    refresh();
    endwin();
}
```

The output for this program would appear as follows:



```

                                     Hello UNIX System User.
$
```

delch()

wdelch(win)

mvdelch(y,x)

mvwdelch(win,y,x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. The rightmost character on the line is set to an unhighlighted blank. This does not necessarily imply use of the hardware delete character feature.

insch(c)

winsch(win, c)

mvinsch(y,x,c)

mvwinsch(win,y,x,c)

The character **c** is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not necessarily imply use of the hardware insert character feature.

inch()

winch(win)

mvinch(y,x)

mvwinch(win,y,x)

The character at the current position in the named window is returned. If any attributes are set for that position, their values will be OR-ed in with the value returned. The predefined constants *A_ATTRIBUTES* and *A_CHARTEXT* can be used with the "&" operator to extract the character or attributes alone.

Video Attributes

The function **addch** always draws two things on a window. In addition to the character itself, a set of *attributes* is associated with the character. These attributes cover various forms of highlighting the character. For example, the character can be put in reverse video, bold, or be underlined. You can think of the attributes as the color of the ink used to draw the character.

A window always has a set of *current attributes* associated with it. The current attributes are associated with each character as it is written to the window. Attributes are a property of the character and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen. The current attributes can be changed with a call to **attrset(attrs)**. (Think of this as dipping the pen for that window in a particular color ink.)

The names of the attributes are:

A_STANDOUT	A_UNDERLINE
A_REVERSE	A_BOLD
A_BLINK	A_BLANK
A_PROTECT	A_ALTCHARSET
A_DIM	

Not all terminals are capable of displaying all attributes. If a particular terminal cannot display a requested attribute, **curses** will attempt to find a substitute attribute. If none is possible, the attribute is ignored.

One particular attribute is called **STANDOUT**. This attribute is used to make text attract the attention of the user. The particular hardware attribute used for standout varies from terminal to terminal and is usually the most visually pleasing attribute the terminal can produce. Standout is typically implemented as reverse video or bold. Many programs do not really need a specific attribute, such as bold or inverse video, but instead just need to highlight some text. For such applications, the **A_STANDOUT** attribute is recommended. Two convenient functions, **standout()** and **standend()** turn on and off this attribute.

Attributes can be turned on in combination. Thus, to turn on blinking bold text, use **attrset(A_BLINK|A_BOLD)**. Individual attributes can be turned on and off with **attron** and **attroff** without affecting other attributes.

For an example program using attributes, see the **highlight** program in Appendix B. The **highlight** program comes about as close to being a filter as is possible with **curses**. It is not a true filter because **curses** must “take over” the terminal screen.

The following functions set the “current attributes” of the named window. These constants are defined in *curses.h* and can be combined with the C | (or) operator.

attrset(at)

wattrset(win, attrs)

The **attrset(at)** function sets the current attributes of the given window to **at**.

attroff(at)

wattroff(win, attrs)

The **attroff(at)** function turns off the named attributes without affecting any other attributes.

attron(at)

wattron(win, attrs)

The **attron(at)** function turns on the named attributes without affecting any others.

standout()

standend()

wstandout(win)

wstandend(win)

The **standout** function is the same as **attron(A_STANDOUT)**.

The **standend** function turns off all attributes, the same as **attrset (0)**.

The following sample shows the use of video attributes. Note that this program is expanded from the sample program shown in the “Writing a String” section presented earlier in this chapter.

```
#include <curses.h>

main( )
{
    initscr();
    attron( A_STANDOUT );
    mvaddstr( LINES / 2, ( COLS / 2 ) - 11, "Hello " );
    attron( A_BLINK );
    refresh();
    printw ( "UNIX " );
    refresh();
    attroff( A_STANDOUT | A_BLINK );
    attron( A_UNDERLINE );
    printw ( "System " );
    refresh();
    attroff( A_UNDERLINE );
    printw ( "Users" );
    refresh();
    endwin();
}
```

If your terminal has all of these attribute capabilities, the output will appear as explained below:

- The word “Hello” will be displayed in *inverse-video*.
- The word “UNIX” will be displayed in *inverse-video* and will also blink.
- The word “System” will return to the normal mode of display, but will be underlined.
- The word “Users” will appear as normal text for your terminal.

Input

Curses can do more than just draw on the screen. Functions are also provided for input from the keyboard. The primary function is **getch()** which waits for the user to type a character on the keyboard and then returns that character. This function is like **getchar** except that it goes through **curses**. Its use is recommended

for programs using the **cbreak()** or **noecho()** options since several terminal or system dependent options become available that cannot be written portably with **getchar**.

Options available with **getch** include **keypad** which allows extra keys such as arrow keys, function keys, and other special keys that transmit escape sequences to be treated as just another key. The values for these keys are over octal 400; so, they should be stored in a variable larger than a **char**. (See the **curses** manual page in Appendix A for a list of function keys and their value.) The **nodelay** mode causes the value **-1** to be returned if there is no input waiting. (Refer to the "Option Setting" heading in the "TERMINAL SETTINGS" section of this chapter for the description of **nodelay**.) Normally, **getch** will wait until a character is typed. Finally, the routine **getstr(str)** can be called, allowing input of an entire line, up to a newline. This routine handles echoing and the erase and kill characters of the user.

See the program **show** in Appendix B for an example use of **getch**.

getch()
wgetch(win)
mvgetch(y,x)
mvwgetch(win,y,x)

A character is read from the terminal associated with the window. In **nodelay** mode, if there is no input waiting, the value **-1** is returned. In **delay** mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**, this will be after one character or after the first newline.

If **keypad** mode is enabled and a function key is pressed, the code for that function key will be returned instead of the raw characters. Possible function keys are defined with integers beginning with 0401 whose names begin with **KEY_**. (Refer to the **curses** manual page in Appendix A.) If a character is received that could be the beginning of a function key (such as escape), **curses** will set a 1-second timer. If the remainder of the sequence does not come in within 1 second, the character will be passed through; otherwise, the function key value will be returned. For this reason, on many terminals there will be a

one second delay after a user presses the escape key. (Using the escape key for a single character function is discouraged.)

getstr(str)
wgetstr(win, str),
mvgetstr(y, x, str)
mvwgetstr(win, y, x, str)

A series of calls to **getch** is made until a newline is received. The resulting value is placed in the area designated by the character pointer **str**. The user's erase and kill characters are interpreted.

scanw(fmt, args)
wscanw(win, fmt, args)
mvscanw(y, x, fmt, args)
mvwscanw(win, y, x, fmt, args)

This function corresponds to **scanf**. The **wgetstr** function is called on the window, and the resulting line is used as input for the scan.

Delays

These functions are not considered to be portable, but are often needed by programs, especially real time response programs, that use **curses**. Some of these functions require a particular operating system or a modification to the operating system to work. In all cases, the routine will compile and return an error status if the requested action is not possible. It is recommended that programmers avoid use of these functions if possible.

draino(0)

The program is suspended until the output queue has drained enough to complete in **0** additional milliseconds. The purpose of this routine is to keep the program (and thus the keyboard) from getting ahead of the screen. If the operating system does not support the ioctls needed to implement **draino**, the value **ERR** is returned; otherwise, **OK** is returned.

napms(ms)

This function suspends the program for **ms** milliseconds. It is similar to **sleep** except with higher resolution. The resolution

actually provided will vary with the facilities available in the operating system, and often a change to the operating system will be necessary to produce good results. The best resolution possible is about .1 seconds. If this resolution is not obtainable, the **napms** routine will round to the next higher second, call **sleep**, and return ERR. Otherwise, the value "OK" is returned.

TERMINAL SETTINGS

Terminal Mode Setting

These functions are used to set modes in the tty driver. The initial mode usually depends on the setting when the program was called: the initial modes documented here represent the normal situation.

longname()

This function returns a pointer to a static area containing a verbose description of the current terminal. It is defined only after a call to **initscr**, **newterm**, or **setupterm**.

The **longname** function does not need any arguments. It returns a pointer to a static area containing the actual long name of the terminal.

echo()

noecho()

These functions control the way characters typed by the user are echoed. Initially, characters typed are echoed by the tty driver. Authors of many interactive programs prefer to do their own echoing in a controlled area of the screen, or they prefer not to echo at all, so they disable echoing.

nl()

nonl()

These functions control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, **curses** is able to make better use of the linefeed capability, resulting in faster cursor motion. Unless you need to have the RETURN key mapped into NEWLINE, it is recommended that you set **nonl()**, in addition to **cbreak()** and **noecho()**.

scroll(win)

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is **stdscr** and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

cbreak()**nocbreak()**

These two functions put the terminal into and out of **cbreak** mode. In this mode, characters typed by the user are immediately available to the program. When out of this mode, the tty driver will buffer characters typed until newline is typed. Interrupt and flow control characters are unaffected by this mode. Initially, the terminal is not in **cbreak** mode. Most interactive programs using **curses** will set this mode.

raw()**noraw()**

The terminal is placed into or out of raw mode. Raw mode is similar to cbreak mode in that characters typed are immediately passed through to the user program. The differences are that in RAW mode the interrupt and quit characters are passed through uninterpreted instead of generating a signal. RAW mode also causes 8-bit input and output. The behavior of the BREAK key and suspend signal may be different on different systems.

The following sample program combines the **echo** and **noecho** functions with some of the Input/Output functions described earlier in this chapter. The program will not echo (display) what is typed by the user. However, the program will respond with a message according to the character typed. If an "h" is entered, "Hello World" is displayed; a "q" will display "Good Bye" and stop the program. Any other letter entered will display the "Sorry only h ..." message.

```
#include    <curses.h>

main( )
{
    int    g;
    initscr();
    cbreak();
    noecho();
    nonl();
    refresh();

    for ( ;; )
    {
        if ( ( g = getch() ) != 'q' )
        {
            if( g == 'h' )
                printw( "Hello World.\n" );
            else
                printw( "Sorry only h will be accepted.\n" );
            refresh();
        }
        else
        {
            printw( "Good Bye.\n" );
            refresh();
            endwin();
            exit( 0 );
        }
    }
}
```

Option Setting

General

These functions set options within **curses**.

clearok(win,bf)

If set, the next call to **wrefresh** with this window will clear the screen and redraw the entire screen. If **win** is **stdscr**, the next call to **wrefresh** with any window will cause the screen to be cleared. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

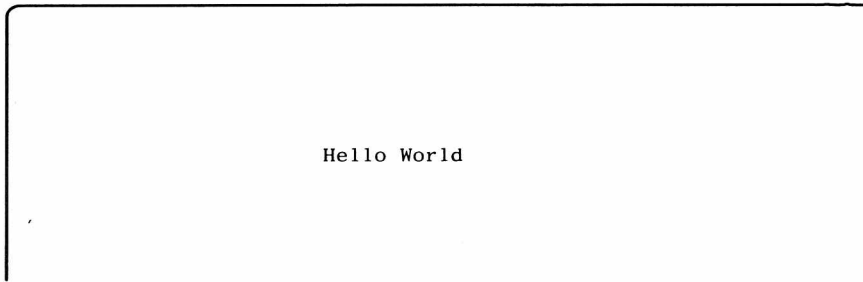
An example of the **clearok** function is shown in the following sample program.

```
#include    < curses.h>

main( )
{
    initscr( );

    move( LINES / 2, ( COLS / 2 ) - 5 );
    printw( "Hello World" );
    refresh( );
    sleep( 1 );
    clearok( stdscr, 1 );
    move( ( LINES / 2 ) + 5, COLS / 3 );
    printw( "Hello Everybody" );
    wrefresh( stdscr );
    endwin( );
}
```

The output from this program will appear in two separate displays. The first display will appear as follows.

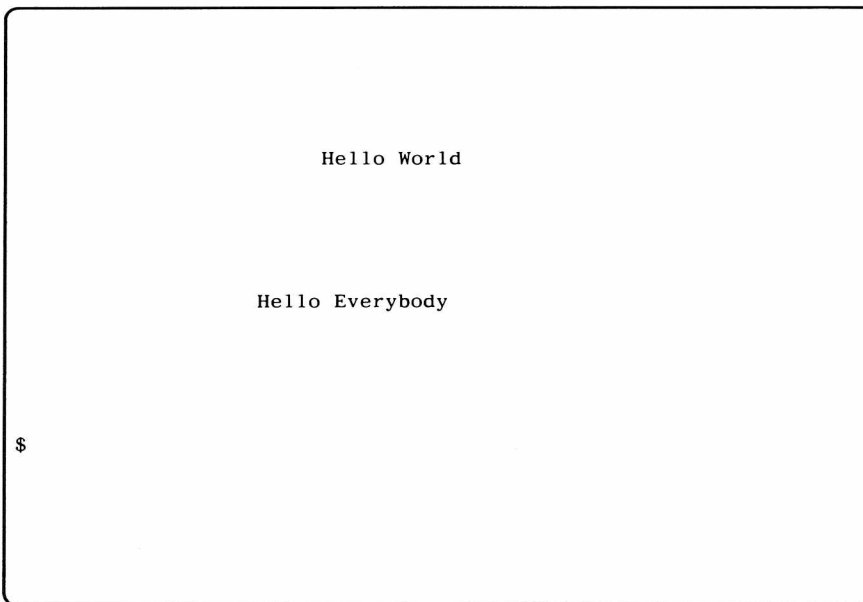


```

Hello World

```

The screen will then be cleared and redrawn with the additional message "Hello Everybody." The resulting display will appear as follows:



```

Hello World

Hello Everybody

$

```

idlok(win,bf)

If enabled, **curses** will consider using the hardware insert/delete line feature of terminals so equipped. If disabled, **curses** will never use this feature. The insert/delete character feature is always considered. Enable this option only if your application needs the insert/delete line, for example, for a

screen editor, or for scrolling. The insert/delete feature is disabled by default because the insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If the insert/delete line cannot be used, **curses** will redraw the changed portions of all lines that do not match the desired line.

keypad(win,bf)

This option enables the keypad of the users terminal. If enabled, the user can press a function key (such as an arrow key), and **getch** will return a single value representing the function key. If disabled, **curses** will not treat function keys specially. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will turn on the terminal keypad.

scrollok(win,bf)

This option controls what happens when the cursor of a window is moved off the edge of the window either from a newline on the bottom line or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, **wrefresh** is called on the window, and then the physical terminal and window are scrolled up one line. Note that in order to get the physical scrolling effect on the terminal, it is also necessary to call **idlok**.

Advanced**leaveok(win,bf)**

Normally, the hardware cursor is left at the location of the window cursor being refreshed. The **leaveok** option allows the cursor to be left wherever the update happens to leave it. The **leaveok** option is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

nodelay(win,bf)

This option causes **getch** to be a nonblocking call. If no input is ready, **getch** will return -1. If disabled, **getch** will hang until a key is pressed.

unctrl(*ch*)

The **unctrl(*ch*)** function is actually a macro which makes a printable representation of the character *ch*. Control characters are displayed in the “^X” notation and printing characters are displayed as is.

Expert

meta(*win*,*bf*)

If enabled, characters returned by **getch** are transmitted with all 8 bits instead of stripping the highest bit. The value **OK** is returned if the request succeeded, the value **ERR** is returned if the terminal or system is not capable of 8-bit input.

Meta mode is useful for extending the non-text command set in applications where the terminal has a meta shift key. Curses takes whatever measures are necessary to arrange for 8-bit input. On some other versions of UNIX Systems, the raw mode will be used. On the 3B2 Computer, the character size will be set to 8, parity checking disabled, and stripping of the 8th bit turned off.

Note that 8-bit input is a fragile mode. Many programs and networks only pass 7 bits. If any link in the chain from the terminal to the application program strips the 8th bit, 8-bit input is impossible.

intrflush(*win*,*bf*)

If this option is enabled when an interrupt key is pressed on the keyboard (interrupt, quit, suspend), all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt but causing **curses** to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default is for the option to be enabled. This option depends on support in the underlying tty driver.

typeahead(*fd*)

Sets the file descriptor for typeahead check. The **fd** variable should be an integer returned from **open** or **fileno**. Setting typeahead to the default value of -1 will disable typeahead

check. By default, file descriptor 0 (stdin) is used. Typeahead is checked independently for each screen, and for multiple interactive terminals it should probably be set to the appropriate input for each screen. A call to **typeahead** always affects only the current screen.

setscrreg(t,b)**wsetscrreg(win,t,b)**

These functions allow the user to set a software scrolling region in a window **win** or **stdscr**. The **t** and **b** variables are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and **scrollok** are enabled, an attempt to move off the bottom margin line (**addch** a newline) will cause all lines in the scrolling region to scroll up one line. Note that this has nothing to do with use of a physical scrolling region capability in the terminal. Only the text of the window is scrolled. If **idlok** is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.

The following sample program shows the usage of some of the more complicated "Option Setting" functions. This program is designed to display several lines separated by a scrolling region. The program will then display lines inside the scrolling region and show the scrolling action as lines are added to the bottom of the region.

SCREEN MANIPULATION

```
#include    <curses.h>

main( )
{
    int      i, t, b;

    initscr();
    t = LINES/3;
    b = ( LINES / 3 ) * 2;
    scrollok( stdscr, TRUE );
    idlok( stdscr, TRUE );
    setscrreg( t, b );
    mvaddstr( t-1, 0, "THIS IS LINE 1 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+1, 0, "THIS IS LINE 2 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+2, 0, "THIS IS LINE 3 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+3, 0, "THIS IS LINE 4 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+4, 0, "THIS IS LINE 5 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+5, 0, "THIS IS LINE 6 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+6, 0, "THIS IS LINE 7 OUTSIDE THE SCROLLING REGION" );
    mvaddstr( b+7, 0, "THIS IS LINE 8 OUTSIDE THE SCROLLING REGION" );
    refresh();
    move( t, 0 );
    for( i = t; i < b+3; i++ )
    {
        printw( "This is line %d of the standard screen.\n", i );
        sleep( 2 );
        refresh();
    }
    endwin();
}
```


Chapter 3

WINDOW MANIPULATION

	PAGE
INITIALIZATION	3-1
General	3-1
Multiple Windows	3-5
INPUT/OUTPUT FUNCTIONS	3-8
General	3-8
Input	3-11
PAD MANIPULATION	3-12

Chapter 3

WINDOW MANIPULATION

INITIALIZATION

General

These functions are some of the more common window manipulation routines. These functions are used to build, move, and delete **curses** windows.

newwin(num_lines, num_cols, beg_row, beg_col)

Create a new window with the given number of lines and columns. The upper left corner of the window is at line **beg_row**, column **beg_col**. If either **num_lines** or **num_cols** is zero, they will be defaulted to **LINES-beg_row** and **COLS-beg_col**. A new full screen window is created by calling **newwin(0,0,0,0)**.

delwin(win)

Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

touchwin(win)

Throw away all optimization information about which parts of the window have been touched by pretending the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window; but the records of which lines have been changed in the other window will not reflect the change.

mvwin(win, br, bc)

Move the window so that the upper left corner will be at position **(br, bc)**. If the move would cause the window to be off the screen, it is an error and the window is not moved.

subwin(orig, num_lines, num_cols, begy, begx)

Create a new window with the given number of lines and columns. The window is at position *begy, begx* on the screen. (It is relative to the screen, not **orig**.) The window is made in the middle of the window **orig**, so that changes made to one window will affect both windows. When using this function, often it will be necessary to call **touchwin** before calling **wrefresh**.

overlay(win1, win2)

overwrite(win1, win2)

These functions overlay **win1** on top of **win2**; that is, all text in **win1** is copied into **win2**. The difference is that **overlay** is nondestructive (blanks are not copied). This means that the bottom window will show through the blank spaces of the top window. The **overwrite** is destructive; that is, the top window will completely cover the bottom window.

The following sample program shows how the **overwrite** function can be used.

```
#include    <curses.h>

main( )
{
    int      j;
    WINDOW   *win1, *win2;

    initscr();
    win1 = newwin( 10, 8, 0, 0 );
    win2 = newwin( 10, 8, 0, 0 );
    for( j=0; j<10; j++ )
        wprintw( win1, "1111111\n" );
    for( j=0; j<10; j++ )
        wprintw( win2, "2 2 2 2\n" );
    wrefresh( win1 );
    wgetch(win1);
    overwrite( win2, win1 );
    wrefresh( win1 );
    endwin();
}
```

The output from this program will appear with two separate windows. The first window will appear as follows:

```
1111111
1111111
1111111
1111111
1111111
1111111
1111111
1111111
1111111
1111111
```

The second window will appear when a character is entered from the keyboard. This window will appear as follows.

```
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
2 2 2 2
```

```
$
```

If this program had been written to show the **overlay** function, the second window would appear as follows:

```
2121212
2121212
2121212
2121212
2121212
2121212
2121212
2121212
2121212
2121212
2121212
```

```
$
```

Note: Notice that the "1's" show through the blank spaces between the "2's" in the output from the **overlay** function.

box(win, vert, hor)

A box is drawn around the edge of the window. The **vert** and **hor** variables are the characters the box is to be drawn with. If **vert** or **hor** have the value of zero, **curses** will substitute reasonable characters for the zero.

refresh()**wrefresh(win)**

This function must be called to get any output on the terminal, as other routines merely manipulate data structures. The **wrefresh** function copies the named window to the physical terminal screen, taking into account what is already there in order to do optimizations. The **refresh** function is the same, using **stdscr** as a default screen. Unless **leaveok** has been enabled, the physical cursor of the terminal is left at the location of the window cursor.

doupdate()**wnoutrefresh(win)**

These two functions allow multiple updates with more efficiency than **wrefresh**. To use them, it is important to understand how **curses** works. In addition to all the window structures, **curses** keeps two data structures representing the terminal screen: a *physical* screen, describing what is actually on the screen, and a *virtual* screen, describing what the programmer *wants* to have on the screen. The **wrefresh** function works by first copying the named window to the virtual screen (**wnoutrefresh**) and then calling the routine to update the screen (**doupdate**). If the programmer wishes to output several windows at once, a series of calls to **wrefresh** will result in alternating calls to **wnoutrefresh** and **doupdate**, causing several bursts of output to the screen. By calling **wnoutrefresh** for each window, it is then possible to call **doupdate** once, resulting in only one burst of output, with probably fewer total characters transmitted.

Multiple Windows

A window is a data structure representing all or part of the terminal screen. It has room for a two dimensional array of characters, attributes for each character (a total of 16 bits per character: 7 for text and 9 for attributes), a cursor, a set of current attributes, and a number of flags. Curses provides a full screen window, called **stdscr**, and a set of functions that use **stdscr**. Another window is provided called **curscr**, representing the physical screen.

It is important to understand that a window is only a data structure. Use of more than one window neither implies use of more than one terminal nor involves more than one process. A window is merely an object which can be copied to all or part of the terminal screen.

The programmer can create additional windows with the **newwin(lines, cols, begin_row, begin_col)** function. This function will return a pointer to a newly created window. The window will be **lines** long by **cols** wide, and the upper left corner of the window will be at screen position (**begin_row, begin_col**). All operations that affect **stdscr** have corresponding functions that affect an arbitrary named window. Generally, these functions have names formed by putting a "w" on the front of the **stdscr** function, and the window name is added as the first parameter. Thus, **waddch(mywin, c)** would write the character **c** to window **mywin**. The **wrefresh(win)** function is used to flush the contents of a window to the screen.

The following sample program shows how to use several of the window manipulation functions. This program will create four different windows on the screen.


```
#include <curses.h>

main()
{
    WINDOW *win1, *win2, *win3, *win4;

    initscr();
    win1 = newwin(LINES/3, COLS, 0, 0);
    win2 = newwin(LINES/3, COLS, LINES/3, 0);
    win3 = newwin(LINES/3, COLS, (LINES/3)*2);
    win4 = newwin(LINES/3, COLS/2, (LINES/3)*2);
    box(win1, '1', '1');
    box(win2, '2', '2');
    box(win3, '3', '3');
    box(win4, '4', '4');
    wrefresh(win1);
    wrefresh(win2);
    wrefresh(win3);
    wrefresh(win4);
    endwin();
    exit(0);
}
```

Windows are useful for maintaining several different screen images, and alternating the user among them. Also, it is possible to subdivide the screen into several windows, refreshing each of them as desired. When windows overlap, the contents of the screen will be the more recently refreshed window. The program **window** in Appendix B is another example of the use of multiple windows.

In all cases, the non-w version of the function calls the w version of the function, using **stdscr** as the additional argument. Thus, a call to **addch(c)** results in a call to **waddch(stdscr, c)**.

For convenience, a set of “move” functions are also provided for most of the common functions. These result in a call to **move** before the other function. For example, **mvaddch(row, col, c)** is the same as **move(row, col); addch(c)**. Combinations like **mvwaddch(row, col, win, c)** also exist.

INPUT/OUTPUT FUNCTIONS

General

The Input/Output functions of “windows” is very similar to the Input/Output functions of “standard” screens (stdscr). The only real difference is that you must also specify the window to receive the action.

wprintw(win, fmt, args)

mvwprintw(win, y, x, fmt, args)

These functions correspond to **printf**. The characters which would be output by **printf** are instead output using **waddch** on the given window.

wmove(win, y, x)

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until **refresh** is called. The position specified is relative to the window. Thus, if you have a window which is not in the upper left corner of the screen, you would have to specify the coordinates as the distance from the upper left corner of the window. The upper left corner of the window is position **(0, 0)**.

werase(win)

This function copies blanks to every position in the window.

wclear(win)

This function is like **werase** but it also calls **clearok**, arranging that the screen will be cleared on the next call to **refresh** for that window.

wclrtoeol(win)

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor is erased.

wclrtoeol(win)

The current line to the right of the cursor is erased.

Inserting and Deleting Text

winsertln(win)

A blank line is inserted above the current line. The bottom line is lost. This does not necessarily imply use of the hardware insert line feature.

wdeleteln(win)

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. This does not necessarily imply use of the hardware delete line feature.

Writing One Character

waddch(win, ch)

mvwaddch(win, y, x, ch)

The character **ch** is put in the window at the current cursor position of the window. If **ch** is a tab, newline, or backspace, the cursor will be moved appropriately in the window. If **ch** is a different control character, it will be drawn in the ^X notation. The position of the window cursor is advanced. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if **scrollok** is enabled, the scrolling region will be scrolled up one line.

The **ch** parameter is actually an integer, not a character. Video attributes can be combined with a character by OR-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another with **inch** and **addch**.)

Writing a String

waddstr(win, str)

mvwaddstr(win, y, x, str)

These functions write all the characters of the null terminated character string **str** on the given window. They are identical to a series of calls to **addch**.

wdelch(win)

mvwdelch(win, y, x)

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position. This does not imply use of the hardware delete character feature.

winsch(win, c)

mvwinsch(win, y, x, c)

The character **c** is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. This does not imply use of the hardware insert character feature.

winch(win)

mvwinch(win, y, x)

The character at the current position in the named window is returned. If any attributes are set for that position, their values will be OR-ed into the value returned. The predefined constants **A_ATTRIBUTES** and **A_CHARTEXT** can be used with the "&" operator to extract the character or attributes alone.

Video Attributes

The video attributes operate inside windows the same as they operate inside screens. The current attribute of the window applies to all characters written into the window and stay with each character through any movement of the character.

wattrset(win, attrs)

The **wattrset(win, at)** function sets the current attributes of the given window to **at**.

wattroff(win, attrs)

The **wattroff(win, at)** function turns off the named attributes (**at**) without affecting any other attributes.

wattron(win, attrs)

The **wattron(win, at)** function turns on the named attributes without affecting any others.

wstandout(win)**wstandend(win)**

The **wstandout** function is the same as **wattron(A_STANDOUT)**. The **wstandend** function is the same as **wattrset(0)**; that is, it turns off all attributes.

Input

wgetch(win)**mvwgetch(win,y,x)**

A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value **-1** is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of **cbreak**, this will be after one character or after the first newline.

Function keys are handled the same in windows as they are in screens. Again, the use of the escape key as a single character function is discouraged.

wgetstr(win,str)**mvwgetstr(win,y,x,str)**

A series of calls to **getch** is made until a newline is received. The resulting value is placed in the area pointed at by the character pointer **str**. The user's erase and kill characters are interpreted.

wscanw(win, fmt, args)**mvwscanw(win, y, x, fmt, args)**

This function corresponds to **scanf**. The **wgetstr** function is called on the window, and the resulting line is used as input for the scan.

getyx(win,y,x)

The cursor position of the window is placed in the two integer variables **y** and **x**. Since this is a macro, no "&" is necessary.

PAD MANIPULATION

A pad is like a window, except that it is not restricted by the screen size, and a pad is not associated with a particular part of the screen. Pads can be used when a large window is needed and only a part of the window will be on the screen at one time.

newpad(num_lines, num_cols)

Creates a new *pad* data structure. Automatic refreshes of pads (for example, scrolling or echoing of input) do not occur. It is not legal to call **refresh** with a pad as an argument; the routines **prefresh** or **pnoutrefresh** should be called instead.

Note that the following routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display. The **prefresh** or **pnoutrefresh** function should be called instead.

prefresh(pad,pminrow,pmincol,sminrow,smincol,smaxrow,smaxcol)

pnoutrefresh(pad,pminrow,pmincol,sminrow,smincol,smaxrow,smaxcol)

These routines are analogous to **wrefresh** and **wnoutrefresh** except that pads are involved instead of windows. The additional parameters are needed to indicate what part of the pad and screen are involved. The **pminrow** and **pmincol** variables specify the upper left corner in the pad of the rectangle to be displayed. The **sminrow**, **smincol**, **smaxrow**, and **smaxcol** variables specify the screen coordinates which define the boundaries (edges) of the rectangle to be displayed. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

The pad functions are very similar to the window functions. The **doupdate** function is used to update the screen like it is used with window structures. The **delwin** function is used to delete a specific pad.

The following sample program shows how to use some of the “pad” functions. This program will output a pad defined by the maximum and minimum cursor positions given on the **prefresh** execution line.

```
#include    <curses.h>

main( )
{
    int      i;

    WINDOW   *pad;

    initscr();
    pad = newpad( 100, 80 );
    for( i=0; i<99; i++ )
        wprintw( pad, "This is line %d of the pad\n", i );
    prefresh( pad, 50, 0, 10, 20, 15, 35 );
    endwin();
}
```

The output of this program would appear as follows.

```
This is line 50  
This is line 51  
This is line 52  
This is line 53  
This is line 54  
This is line 55
```

\$

Note: This output may not be shown to scale. The output on your terminal will begin 10 lines down and 20 spaces from the left margin.

Chapter 4

MULTIPLE TERMINALS

	PAGE
GENERAL PROGRAM FORMAT	4-2
FUNCTIONS	4-3

Chapter 4

MULTIPLE TERMINALS

Curses can produce output on more than one terminal at once. This is useful for single process programs that access a common database such as multi-player games. Output to multiple terminals is a difficult business, and **curses** does not solve all the problems for the programmer. It is the responsibility of the program to determine the file name of each terminal line and what kind of terminal is on each of those lines. The standard method, checking **\$TERM** in the environment, does not work since each process can only examine its own environment.

Another problem that must be solved is that of multiple programs reading from one line. This situation produces a race condition and should be avoided. Nonetheless, a program wishing to take over another terminal cannot just shut off whatever program is currently running on that line. (Usually, security reasons would also make this inappropriate. However, for some applications such as an inter-terminal communication program or a program that takes over unused tty lines, shutting off other programs would be appropriate.) A typical solution requires the user logged in on each line to run a program notifying the master program that the user is interested in joining the master program and telling it the notification program process ID, the name of the tty line, and the type of terminal being used. Then, the program goes to sleep until the master program

finishes. When done, the master program wakes up the notification program, and all programs exit.

Curses handles multiple terminals by always having a “current terminal.” All function calls always affect the current terminal. The master program should set up each terminal, saving a reference to the terminals in its own variables. When the master program wishes to affect a terminal, it should set the current terminal as desired, and then call ordinary **curses** routines.

References to terminals are of the type **struct screen ***. A new terminal is initialized by calling **newterm(type, outfd, infd)**. The **newterm** function returns a screen reference to the terminal being set up. The **type** variable represents a character string, naming the kind of terminal being used. The **outfd** and **infd** variables are *stdio* file descriptors to be used for input and output to the terminal. (If only output is needed, the file can be opened for output only.) This call replaces the normal call to **initscr**, which calls **newterm(getenv("TERM"), stdout, stdin)**.

To change the current terminal, call “**set_term(sp)**” where **sp** is the screen reference to be made current. The **set_term** function returns a reference to the previous terminal.

GENERAL PROGRAM FORMAT

It is important to realize that each terminal has its own set of windows and options. Each terminal must be initialized separately with **newterm**. Options such as **cbreak** and **noecho** must be set separately for each terminal. The functions **endwin** and **refresh** must be called separately for each terminal. The following sample program is a typical scenario to output a message to each terminal.

```
for (i=0; i<nterm; i++) {  
    set_term(terms[i]);  
    mvaddstr(0, 0, " Important message");  
    refresh();  
}
```

See the sample program **two** in Appendix B for a full example. This program pages through a file, showing one page to the first terminal and the next page to the second terminal. Since no standard multiplexor is available in current versions of the UNIX System, it is necessary to either busy wait or call **sleep(1)**; between each check for keyboard input. The **two** program sleeps for a second between checks.

The **two** program is just a simple example of two terminal **curses**. It does not handle notification, as described above, instead it requires the name and type of the second terminal on the command line. As written, the command **sleep 100000** must be typed on the second terminal to put it to sleep while the program runs, and the first user must have both read and write permission on the second terminal.

FUNCTIONS

resetty()

savetty()

These functions save and restore the state of the tty modes.

The **savetty** function saves the current state in a buffer, **resetty** restores the state to what it was at the last call to **savetty**.

newterm(type, fd)

A program which outputs to more than one terminal should use **newterm** instead of **initscr**. The **newterm** function should be called once for each terminal. It returns a variable of type **SCREEN *** which should be saved as a reference to that terminal. The arguments are the type of the terminal (a string) and a stdio file descriptor (**FILE ***) for output to the terminal. The file descriptor should be open for both reading and writing if input from the terminal is desired. The program should also

call **endwin** for each terminal being used (see **set_term** below). If an error occurs, the value **NULL** is returned.

set_term(new)

This function is used to switch to a different terminal. The screen reference **new** becomes the new current terminal. The previous terminal is returned by the function. All other calls affect only the current terminal.

Chapter 5
PORTABILITY FUNCTIONS

	PAGE
FUNCTIONS	5-1

Chapter 5

PORTABILITY FUNCTIONS

These functions do not directly involve terminal dependent character output but tend to be needed by programs that use **curses**. Unfortunately, their implementation varies from one version of the UNIX Operating System to another.

FUNCTIONS

The following functions have been included here to enhance the portability of programs using **curses**.

erasechar()

The erase character chosen by the user is returned to the program. This is the character typed by the user to erase the character just typed.

killchar()

The line kill character chosen by the user is returned to the program. This is the character typed by the user to forget the entire line being typed.

flushinp()

The **flushinp()** instruction throws away any typeahead that has been typed by the user and has not yet been read by the program.

baudrate()

The **baudrate()** instruction returns the output speed of the terminal. The number returned is the integer baud rate, for example, 9600 rather than a table index such as B9600. This function can also be used to check the status of a terminal. If a "0" is returned, the terminal being checked is "off-line." This may suggest a reset (**resetty**) for that terminal if the terminal is supposed to be operating, or it may mean the terminal has been turned OFF.

Chapter 6

LOWER LEVEL FUNCTIONS

	PAGE
LOW LEVEL TERMINFO USAGE	6-1
Cursor Motion	6-4
Terminfo Level	6-4

Chapter 6

LOWER LEVEL FUNCTIONS

LOW LEVEL TERMINFO USAGE

Some programs need to use lower level primitives than those offered by **curses**. For such programs, the *terminfo level* interface is offered. This interface does not manage your terminal screen, but rather gives you access to strings and capabilities which you can use yourself to manipulate the terminal.

Programmers are discouraged from using this level. Whenever possible, the higher level **curses** routines should be used. This will make your program more portable to other UNIX Systems and to a wider class of terminals. **Curses** takes care of all the glitches and misfeatures present in physical terminals; but at the terminfo level, you must deal with them yourself. Also, you cannot be guaranteed that this part of the interface will not change or be upward compatible with previous releases.

There are two circumstances when it is proper to use terminfo. The first is when you are writing a special purpose tool that sends a special purpose string to the terminal, such as programming a function key, setting tab stops, sending output to a printer port, or dealing with the status line. The second situation is when writing a filter. A typical filter does one transformation on the input stream

without clearing the screen or addressing the cursor. If this transformation is terminal dependent and clearing the screen is inappropriate, use of terminfo is indicated.

The following is a typical format for a program written at the terminfo level.

```
#include <curses.h>
#include <term.h>
...
    setupterm(0, 1, 0);
    ...
    putp(clear_screen);
    ...
    reset_shell_mode( );
    exit(0);
```

Initialization is done by calling **setupterm**. Passing the values 0, 1, and 0 invokes reasonable defaults. If **setupterm** cannot figure out what kind of terminal you are on, it will print an error message and exit. The program should call **reset_shell_mode** before it exits.

Global variables with names like **clear_screen** and **cursor_address** are defined by the call to **setupterm**. (See the **terminfo** manual page in Appendix A for a complete list of capabilities.) They can be output using **putp** or **tputs** which allows the programmer more control. These strings should not be directly output to the terminal using **printf** since they contain padding information. A program that directly outputs strings will fail on terminals that require padding or that use the xon/xoff flow control protocol.

In the terminfo level, the higher level routines described previously are not available. It is up to the programmer to output whatever is needed. For a list of capabilities and a description of what they do, see the **terminfo** manual page found in Appendix A.

The example program **termhl** in Appendix B shows a simple use of terminfo. It is a version of **highlight** that uses **terminfo** instead of

curses. This version can be used as a filter. The strings to enter bold and underline mode, and to turn off all attributes, are used.

The **termhl** program is more complex than it need be in order to illustrate some properties of terminfo. The routine **vidattr** could have been used instead of directly outputting **enter_bold_mode**, **enter_underline_mode**, and **exit_attribute_mode**. In fact, the program would be more robust if it used **vidattr** since there are several ways to change video attribute modes. This program was written to illustrate typical use of terminfo.

The function **tputs(cap, affcnt, outc)** applies padding information. Some capabilities contain strings like **\$<20>**, which means to pad for 20 milliseconds. The **tputs** command generates enough pad characters to delay for the appropriate time. The first parameter is the string capability to be output. The second is the number of lines affected by the capability. (Some capabilities may require padding that depends on the number of lines affected. For example, **insert_line** may have to copy all lines below the current line and may require time proportional to the number of lines copied. By convention, **affcnt** is the value 1 rather than 0 if no lines are affected. The value 1 is used for safety reasons, since **affcnt** is multiplied by the amount of time per item, and anything multiplied by 0 is 0.) The third parameter is a routine to be called with each character.

For many simple programs, **affcnt** is always 1 and **outc** always just calls **putchar**. For these programs, the routine **putp(cap)** is a convenient abbreviation. The **termhl** example could be simplified by using **putp**.

Note also the special check for the **underline_char** capability. Some terminals, rather than having a code to start underlining and a code to stop underlining, have a code to underline the current character. The **termhl** program keeps track of the current mode; and if the current character is supposed to be underlined, the program will output **underline_char** if necessary. Low level details such as this are precisely why the **curses** level is recommended over the **terminfo** level. Curses takes care of terminals with different methods of underlining and other terminal functions.

Cursor Motion

mvcur(oldrow, oldcol, newrow, newcol)

This routine optimally moves the cursor from (oldrow, oldcol) to (newrow, newcol). The user program is expected to keep track of the current cursor position. Note that unless a full screen image is kept, **curses** will have to make pessimistic assumptions, sometimes resulting in less than optimal cursor motion. For example, moving the cursor a few spaces to the right can be done by transmitting the characters being moved over; but if **curses** does not have access to the screen image, it doesn't know what these characters are.

Terminfo Level

These routines are called by low level programs that need access to specific capabilities of **terminfo**. A program working at this level should include both *curses.h* and *term.h*, in that order. After a call to **setupterm**, the capabilities will be available with macro names defined in *term.h*. See the **terminfo** manual page in Appendix A for a detailed description of the capabilities.

Boolean valued capabilities will have the value 1 if the capability is present — the value 0 if it is not. Numeric capabilities have the value -1 if the capability is missing and have a value of at least 0 if it is present. String capabilities (both those with and without parameters) have the value **NULL** if the capability is missing; otherwise, they have type **char *** and point to a character string containing the capability. The special character codes involving the **** and **^** characters (such as **\r** for return, or **^A** for control A) are translated into the appropriate American Standard Code for Information Interchange (ASCII) characters. Padding information (of the form **\$<time>**) and parameter information (beginning with **%**) are left uninterpreted at this stage. The routine **tputs** interprets padding information, and **tparm** interprets parameter information.

If the program only needs to handle one terminal, the definition **-DSINGLE** can be passed to the C Language compiler resulting in static references to capabilities instead of dynamic references. This can result in smaller program code, but it prevents use of more than

one terminal at a time. Very few programs use more than one terminal; so, almost all programs can use this flag.

setupterm(term, filenum, errret)

This routine is called to initialize a terminal. The **term** variable represents the character string representing the name of the terminal being used. The **filenum** variable is the UNIX System file descriptor of the terminal being used for output. The **errret** variable is a pointer to an integer in which a success or failure indication is returned. The values returned can be 1 (all is well), 0 (no such terminal), or -1 (some problem locating the **terminfo** database).

The value of **term** can be given as 0, which will cause the value of TERM in the environment to be used. The **errret** pointer can also be given as 0, meaning no error code is wanted. If **errret** is defaulted and something goes wrong, **setupterm** will print an appropriate error message and exit rather than returning. Thus, a simple program can call **setupterm(0, 1, 0)** and not worry about initialization errors.

If the variable TERMINFO is set in the environment to a path name, **setupterm** will check for a compiled **terminfo** description of the terminal under that path before checking */usr/lib/terminfo/*/**. Otherwise, only */usr/lib/terminfo/*/** is checked.

The **setupterm** function will check the tty driver mode bits, using **filenum**, and change any that might prevent the correct operation of other low-level routines. Currently, the mode that expands tabs into spaces is disabled because the tab character is sometimes used for different functions by different terminals. (Some terminals use it to move right one space. Others use it to address the cursor to row or column 9.) If the system is expanding tabs, **setupterm** will remove the definition of the **tab** and **backtab** functions, making the assumption that since the user is not using hardware tabs, they may not be properly set in the terminal. Other system dependent changes, such as disabling a virtual terminal driver, may be made here.

As a side effect, **setupterm** initializes the global variable **ttytype**, which is an array of characters, to the value of the list of names for the terminal. This list comes from the beginning of the **terminfo** description.

After the call to **setupterm**, the global variable **cur_term** is set to point to the current structure of terminal capabilities. By calling **setupterm** for each terminal and saving and restoring **cur_term**, it is possible for a program to use two or more terminals at once.

The mode that turns newlines into Carriage Return Line Feed (CRLF) on output is not disabled. Programs that use **cursor_down** or **scroll_forward** should disable this mode if their value is linefeed. The **setupterm** function calls **reset_prog_mode** after any changes it makes.

reset_prog_mode()
reset_shell_mode()
def_prog_mode()
def_shell_mode()

These routines can be used to change the tty modes between the two states: *shell* (the mode they were in before the program was started) and *program* (the mode needed by the program). The **def_prog_mode** function saves the current terminal mode as program mode. The **setupterm** function and **initscr** call **def_shell_mode** automatically. The **reset_prog_mode** function puts the terminal into program mode, and **reset_shell_mode** puts the terminal into normal mode. These functions set the tty driver only, they do not transmit anything to the terminal.

A typical calling sequence is for a program to call **initscr** (or **setupterm** if a **terminfo** level program), then to set the desired program mode by calling routines such as **cbreak** and **noecho**, then to call **def_prog_mode** to save the current state. Before a shell escape or control-Z suspension, the program should call **reset_shell_mode** to restore normal mode for the shell. Then, when the program resumes, it should call **reset_prog_mode**. Also, all programs must call **reset_shell_mode** before they exit. (The higher level routine **endwin** automatically calls **reset_shell_mode**.)

Normal mode is stored in **cur_term->Ottyb**, and program mode is stored in **cur_term->Nttyb**. These structures are both of type **SGTTYB** (which varies depending on the system). Currently, the possible types are **struct sgtyb** (on some other systems) and **struct termio** (on this version of the UNIX System). The **def_prog_mode** function should be called to save the current state in **Nttyb**.

vidputs(newmode, putc)

The **newmode** variable is any combination of attributes, defined in *curses.h*. The **putc** variable is a "putchar-like" function. The proper string to put the terminal in the given video mode is output. The previous mode is remembered by this routine. The result characters are passed through **putc**.

vidattr(newmode)

The proper string to put the terminal in the given video mode is output to **stdout**.

tparm(instring, p1, p2, p3, p4, p5, p6, p7, p8, p9)

The **tparm** function is used to instantiate a parameterized string. The character string returned has the given parameters applied and is suitable for **tputs**. Up to 9 parameters can be passed in addition to the parameterized string.

tputs(cp, affcnt, outc)

A string capability, possibly containing padding information, is processed. Enough padding characters to delay for the specified time replace the padding specification, and the resulting string is passed one character at a time to the routine **outc** which should expect one character as a parameter. (This routine often just calls **putchar**.) The **cp** variable is the capability string. The **affcnt** variable is the number of units affected by the capability, which varies with the particular capability. (For example, the **affcnt** for **insert_line** is the number of lines below the inserted line on the screen, that is, the number of lines that will have to be moved by the terminal.) The **affcnt** value is used by the padding information of some terminals as a multiplication factor. If the capability does not have a factor, the value 1 should be passed.

putp(str)

This is a convenient function to output a capability with no **affcnt**. The string is output to **putchar** with an **affcnt** value of 1. It can be used in simple applications that do not need to process the output of **tputs**.

delay_output(ms)

A delay is inserted into the output stream for the given number of milliseconds. The current implementation inserts sufficient pad characters for the delay. This should not be used in place of a high resolution sleep, but rather for delay effects in the output. Due to buffering in the system, it is unlikely that this call will result in the process actually sleeping. Since large numbers of pad characters can be output, it is recommended that **ms** not exceed 500.

Chapter 7

TERMINFO DATABASE

	PAGE
PREPARING DESCRIPTIONS	7-1
Naming the Terminal	7-2
Defining Capabilities	7-3
Compiling the New Entry	7-9
Testing an Entry	7-10
TPUT COMMAND	7-11
TERMCAP AND TERMINFO COMPATIBILITY	7-13

Chapter 7

TERMINFO DATABASE

The **terminfo** database describes terminals by giving a set of capabilities and by describing how the terminal performs certain operations. Each terminal description contains the names by which the terminal is known and a group of comma separated fields describing the actions and capabilities of the terminal.

PREPARING DESCRIPTIONS

If there is no terminal description for your terminal, the entry must be built from scratch. You may wish to use partial descriptions and test them as you go along. These tests may expose deficiencies in the ability to describe the terminal. The general procedure for building this description is as follows:

1. Give the known names of the terminal.
2. List and define the known capabilities.
3. Compile the newly created description entry.
4. Test the entry for correct operation.

Naming the Terminal

The name of the terminal is the first information given in each description. This string of names, assuming there is more than one name, is separated by "pipe" symbols (`|`). The first name given should be the most common abbreviation for the terminal. The last name given should be a long name fully identifying the terminal. It is usually the manufacturer's formal name for the terminal. All names between the first and last entries should be known synonyms for the terminal name. All names but the formal name should be typed in lowercase letters and contain no blanks. Naturally, the formal name is entered as closely as possible to the manufacturer's name.

Terminal names should follow common naming conventions. These conventions start with a root name; *myterm*, for example. The root name should not contain odd characters, like hyphens, that may not be recognized as a synonym for the terminal name. Possible hardware modes or user preferences should be indicated by adding a hyphen and the mode indicator at the end of the name. For example, "wide mode" which is indicated by a `-w` would be given as "*myterm-w*." The following suffixes should be used whenever possible:

Suffix	Meaning	Example
<code>-w</code>	Wide mode (more than 80 columns)	<i>myterm-w</i>
<code>-am</code>	Automatic margins (usually default)	<i>myterm-am</i>
<code>-nam</code>	No automatic margins	<i>myterm-nam</i>
<code>-n</code>	Number of lines on the screen (This example defines 30 lines.)	<i>myterm-30</i>
<code>-xm</code>	Number of columns	<i>myterm-x132</i>
<code>-nxm</code>	Both number of lines and columns	<i>myterm-30x80</i>
<code>-na</code>	No arrow keys except in local mode	<i>myterm-na</i>
<code>-np</code>	Number of pages in memory (This example defines 4 pages.)	<i>myterm-4p</i>

-rv	Sets reverse video	myterm-rv
-s	With optional status line enabled	myterm-s

The following example is the name string from the description of the TELETYPE* 5420 Buffered Display Terminal.

5420|tty5620|teletype 5420,

Defining Capabilities

The capabilities for each terminal are described in a string of comma separated fields. This string of fields may continue onto multiple lines as long as white space (that is; tabs, spaces) begins each line except the first line of each description. Comments can be included in the description by entering a number symbol (#) at the beginning of the line.

A complete list of the **terminfo** capabilities is given in the **terminfo** manual page found in Appendix A. This list contains the name of the capability, the abbreviated name used in the database, the two-letter code that corresponds to the old **termcap** database name, and a short description of the capability. The abbreviated name that you will use in your database descriptions is shown in the column titled **Capname**.

Each terminal description contains abbreviated names of capabilities. Some capabilities also require a terminal-specific instruction which performs the named capability. For example, **bel** is the abbreviated name for the beeping or ringing capability. On most terminals, a "control-g" (^G) is the instruction that produces a beeping sound. Therefore, the beeping capability would be shown in the terminal description as "**bel=^G,.**"

* Registered trademark of AT&T Teletype Corporation

The terminal-specific instruction can be a keyed operation (like “^G”), a numeric value, or a parameter string containing the sequence of operations required to achieve the particular capability. There are certain characters which are used after the capability name to indicate what type of instruction is required. These characters are explained as follows:

- # Indicates a numeric value is to follow. This character follows a capability which needs a number as the instruction. For example, the number of columns is defined as “**cols#80,**”
- = Indicates that the capability instruction is the character string which follows. This string instructs the terminal how to act and may actually be a sequence of commands. There are certain characters used in the instruction strings which have special meanings. These special characters are explained as follows:
 - ^ Indicates a control character is to be used. For example, the beeping sound is produced by a “control-G.” This would be shown as “^G.”
 - \E or \e These characters followed by another character indicate an ESCAPE instruction. An entry of \EC would transmit to the terminal as “ESCAPE-C.”
 - \n These characters provide a “newline” instruction.
 - \l These characters provide a “linefeed” instruction.
 - \r These characters provide a “return” instruction.
 - \t These characters provide a “tab” instruction.

- \b These characters provide a "backspace" instruction.
- \f These characters provide a "formfeed" instruction.
- \s These characters provide a "space" instruction.
- \$< > These symbols are used to indicate a delay in milliseconds. The desired amount of delay is enclosed inside the "less than/greater than" symbols (< >). The amount of delay may be a whole number, a numeric value to one decimal place (tenths), or a number of either form followed by an asterisk (*). The "*" indicates that the delay will be proportional to the amount of lines affected by the operation. For example, a 20-millisecond delay would appear as "\$<20*>."

Sometimes, it may be necessary to comment out a capability so that the terminal ignores this particular field. This is done by placing a period (.) in front of the abbreviated name for the capability. For example, if you would like to comment out the beeping capability, the description entry would appear as **".bel=^G,."**

Basic Capabilities

To build a description from scratch, you would normally start listing the capabilities immediately below the terminal names. The owner's manual for your terminal should provide information on what capabilities are available and what character string makes up the correct instruction to perform each capability. It may be beneficial to start with those capabilities which are common to almost all terminals. Some of the common traits of all terminals are bells, number of columns, number of lines on the screen, and overstriking of characters if necessary.

For the following example of building a **terminfo** description, we will use a fictitious terminal name. The name string for the terminal is shown as follows:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,
```

Suppose this fictitious terminal has the following capabilities. The **terminfo** manual page of Appendix A lists these capabilities and gives the abbreviated name to use in the database. The appropriate abbreviated name is shown in parentheses immediately after the capability description.

- An automatic wrap around to the beginning of the next line whenever the cursor reaches the right-hand margin. (**am**)
- The ability to produce a beeping sound. The instruction required to produce the beeping sound is “^G.” (**bel**)
- An 80-column wide screen. (**cols**)
- A 30-line long screen. (**lines**)
- An ability to retain the display below the screen. (**db**)

By combining the name string and the capability descriptions that we now have, we can put together a very general **terminfo** database entry. Remember that each field must be separated by a comma. The entry would look like this:

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,  
am, bel=^G, cols#80, lines#30, db,
```

Keyboard Entered Capabilities

The keyboard entered capabilities are those actions which occur when a key is struck on the keyboard. Although the capabilities may be common to many terminals, the instructions to perform the operation could be very different. These instructions are related specifically to the terminal which is being described. For example, a carriage return may be indicated by an “**^M**” (control-M) on one terminal. The indication for a carriage return on another terminal may be an “**\EG**” (ESCAPE-G).

The following characteristics help describe the before mentioned fictitious terminal. Again, the abbreviated command associated with the given capability is shown in parentheses.

- A carriage return is indicated by a control-M. (**cr**)
- A cursor up one line motion is indicated by a control-K. (**cuu1**)
- A cursor down one line motion is indicated by a control-J. (**cud1**)
- Moving the cursor to the left one space is indicated by a control-H. (**cub1**)
- Moving the cursor to the right one space is indicated by a control-L. (**cuf1**)
- Entering reverse video mode is indicated by an ESCAPE-D. (**sms0**)
- Exiting reverse video mode is given by an ESCAPE-Z. (**rmso**)
- A clear to the end of a line instruction is indicated by an ESCAPE-K and should have a 3-millisecond delay. (**el**)

These capabilities must be added to the general description entry of **myterm**. To do this, simply continue the description in the same manner with the new information. The resulting database entry is shown as follows.

```
myterm|mytm|mine|fancy|terminal|My FANCY Terminal,  
    am, bel=^G, cols#80, lines#30, db, cr=^M,  
    cuu1=^K, cud1=^J, cub1=^H, cuf1=^L, smso=\ED,  
    rmso=\EZ, el=\EK$<3>,
```

Note: This is a very short, very simple **terminfo** database entry. It is shown for illustration purposes only. DO NOT attempt to use this example as an actual database entry for any terminal.

There are other capabilities which are described using parameter strings. Some parameter string instructions may be the same for different terminals. For example, terminals which conform to the American National Standards Institute (ANSI) standards for computer terminals will all have the same instruction for cursor address (**cup**) and setting attributes (**sgr**). The use of parameter strings is highly complex. If you have a need to know, you should be able to understand the information on parameter strings given in the **terminfo** manual page of Appendix A.

The procedure for building a terminal description and the example shown in this discussion should be enough to show you how a database entry is constructed. Almost all capability descriptions will be defined in one of the forms shown in the example.

Compiling the New Entry

General

The **terminfo** database entries are compiled using the **tic** compiler. This compiler translates **terminfo** database entries from the source format into the compiled format.

The source file for the description must be in a file suffixed with **.ti**. For example, the fictitious database entry being used for illustration purposes would be in a source file named **myterm.ti**. The compiled version is placed in `/usr/lib/terminfo/*` where `*` is a directory named with the first letter of each entry. For example, the compiled description of **myterm** (source file **myterm.ti**) would be placed in `/usr/lib/terminfo/m` since the first letter in the description entry is "m" (myterm).

Command Format

The general format for the **tic** compiler is as follows:

```
tic [-v] file file2 ...
```

The `-v` option causes the compiler to trace its actions and output information about its progress.

The *file* fields are to indicate which file is to be compiled. Notice that more than one file can be compiled at one time if the filenames are separated by a space.

Sample Command

The following command line shows how to compile a **terminfo** source file named **myterm.ti**. The verbose option (`-v`) is included.

```
$ tic -v myterm.ti<CR>
(The trace information will
 appear when the compilation
 is complete.)
$
```

Note: The <CR> symbol is used to show a carriage return.

Refer to the **tic** manual page in Appendix A for more information on the **terminfo** compiler.

Testing an Entry

You can test a new terminal description entry by setting the environment variable "TERMINFO" to the path name of the directory containing the newly compiled description. If the "TERMINFO" variable is set to a directory before the entry is compiled, the compiled entry will be placed in the "TERMINFO" directory. All programs will look in the new "TERMINFO" directory description file rather than in */usr/lib/terminfo*. If the programs run the same on the new terminal as they did on the older known terminals, then the new description is functional.

A way to test for correct insert line padding is to edit (using *vi*) a rather large file (over 100 lines) at 9600 baud (if possible), and delete about 15 lines from the middle of the screen. Hit "u" (undo) several times quickly. If the terminal messes up, then more padding is usually required. A similar test can be used for inserting a character.

TPUT COMMAND

General

The **tput** command uses the **terminfo** database to output terminal-specific capabilities and other information to the screen or shell. This command outputs a string or an integer according to the type of capability being described. If the capability is a Boolean expression, then **tput** sets the exit code (0 for TRUE, 1 for FALSE) and produces no output.

Command Format

The general format for the **tput** command is as follows:

```
tput [-Ttype] capname
```

The type of terminal you are requesting information about is identified with the *-Ttype* option. Usually, this option is not necessary because the default terminal name is taken from the environment variable **\$TERM**.

The *capname* field is used to indicate which capability to output from the **terminfo** database.

Sample Command

The following command line shows how to output the clear-screen instruction for the terminal being used.

```
$ tput clear<CR>
(The instruction for clear-screen
appears here.)
$
```

The following command line shows how to output the number of columns for the terminal being used.

```
$ tput cols<CR>
(The number of columns used by
the terminal will appear here.)
$
```

The **tput** manual page found in Appendix A contains more information on the usage and possible messages associated with this command.

TERMCAP AND TERMINFO COMPATIBILITY

The **terminfo** database is designed to take the place of the **termcap** database. Due to the many programs and processes that have been written with and for the **termcap** database, it will be impossible to make a complete cutover at one time. There can be programs written to convert the **termcap** description entries into **terminfo** description entries. However, any conversion from **termcap** to **terminfo** requires a great deal of knowledge about both databases. All entrances into the databases should be handled with extreme caution. These files are very important to the operation of your computer.

If you have been using cursor optimization programs with the *-ltermcap* option in the "cc" command line, those programs will still be functional. However, the *-ltermcap* option must be replaced with the *-lcurses* option.

Appendix A

MANUAL PAGES

This appendix contains the UNIX System manual pages for the Terminal Information Utilities. Manual pages for the following commands are provided in alphabetical sequence.

curses	tic
terminfo	tput

You have the capability to arrange the manual pages provided in this guide to support your specific needs. The yellow sheet provided at the front of this guide describes your options for filing the manual pages and descriptive information.

NAME

curses — CRT screen handling and optimization package

SYNOPSIS

```
#include <curses.h>
cc [ flags ] files -lcurses [ libraries ]
```

DESCRIPTION

These routines give the user a method of updating screens with reasonable optimization. In order to initialize the routines, the routine *initscr()* must be called before any of the other routines that deal with windows and screens are used. The routine *endwin()* should be called before exiting. To get character-at-a-time input without echoing, (most interactive, screen oriented-programs want this) after calling *initscr()* you should call "*nonl(); cbreak(); noecho();*"

The full curses interface permits manipulation of data structures called *windows* which can be thought of as two dimensional arrays of characters representing all or part of a CRT screen. A default window called *stdscr* is supplied, and others can be created with *newwin*. Windows are referred to by variables declared "*WINDOW **", the type *WINDOW* is defined in *curses.h* to be a C structure. These data structures are manipulated with functions described below, among which the most basic are *move*, and *addch*. (More general versions of these functions are included with names beginning with 'w', allowing you to specify a window. The routines not beginning with 'w' affect *stdscr*.) Then *refresh()* is called, telling the routines to make the users CRT screen look like *stdscr*.

If the environment variable *TERMINFO* is defined, any program using curses will check for a local terminal definition before checking in the standard place. For example, if the standard place is */usr/lib/terminfo*, and *TERM* is set to "*vt100*", then normally the compiled file is found in */usr/lib/terminfo/v/vt100*. (The "*v*" is copied from the first letter of "*vt100*" to avoid creation of huge directories.) However, if *TERMINFO* is set to */usr/mark/myterms*, curses will first check */usr/mark/myterms/v/vt100*, and if that fails, will then check */usr/lib/terminfo/v/vt100*. This is useful for developing experimental definitions or when write permission in */usr/lib/terminfo* is not available.

SEE ALSO

terminfo(4).

3B2 Computer System Terminal Information Utilities Guide.

FUNCTIONS

Routines listed here may be called when using the full curses.

<i>addch(ch)</i>	add a character to <i>stdscr</i> (like <i>putchar</i>) (wraps to next line at end of line)
<i>addstr(str)</i>	calls <i>addch</i> with each character in <i>str</i>
<i>attroff(attrs)</i>	turn off attributes named
<i>attron(attrs)</i>	turn on attributes named
<i>attrset(attrs)</i>	set current attributes to <i>attrs</i>
<i>baudrate()</i>	current terminal speed
<i>beep()</i>	sound beep on terminal
<i>box(win, vert, hor)</i>	draw a box around edges of <i>win</i> <i>vert</i> and <i>hor</i> are chars to use for <i>vert.</i> and <i>hor.</i> edges of box
<i>clear()</i>	clear <i>stdscr</i>
<i>clearok(win, bf)</i>	clear screen before next redraw of <i>win</i>
<i>clrtoebot()</i>	clear to bottom of <i>stdscr</i>
<i>clrtoeol()</i>	clear to end of line on <i>stdscr</i>
<i>cbreak()</i>	set <i>cbreak</i> mode
<i>delay_output(ms)</i>	insert <i>ms</i> millisecond pause in output

<code>delch()</code>	delete a character
<code>deleteln()</code>	delete a line
<code>delwin(win)</code>	delete <i>win</i>
<code>doupdate()</code>	update screen from all <i>wnooutrefresh</i>
<code>echo()</code>	set echo mode
<code>endwin()</code>	end window modes
<code>erase()</code>	erase <i>stdscr</i>
<code>erasechar()</code>	return user's erase character
<code>fixterm()</code>	restore tty to "in curses" state
<code>flash()</code>	flash screen or beep
<code>flushinp()</code>	throw away any typeahead
<code>getch()</code>	get a char from tty
<code>getstr(str)</code>	get a string through <i>stdscr</i>
<code>gettmode()</code>	establish current tty modes
<code>getyx(win, y, x)</code>	get (y, x) co-ordinates
<code>has_ic()</code>	true if terminal can do insert character
<code>has_il()</code>	true if terminal can do insert line
<code>idlok(win, bf)</code>	use terminal's insert/delete line if <i>bf</i> != 0
<code>inch()</code>	get char at current (y, x) co-ordinates
<code>initscr()</code>	initialize screens
<code>insch(c)</code>	insert a char
<code>insertln()</code>	insert a line
<code>intrflush(win, bf)</code>	interrupts flush output if <i>bf</i> is TRUE
<code>keypad(win, bf)</code>	enable keypad input
<code>killchar()</code>	return current user's kill character
<code>leaveok(win, flag)</code>	OK to leave cursor anywhere after refresh if <i>flag</i> != 0 for <i>win</i> , otherwise cursor must be left at current position.
<code>longname()</code>	return verbose name of terminal
<code>meta(win, flag)</code>	allow meta characters on input if <i>flag</i> != 0
<code>move(y, x)</code>	move to (y, x) on <i>stdscr</i>
<code>mvaddch(y, x, ch)</code>	move(y, x) then <i>addch(ch)</i>
<code>mvaddstr(y, x, str)</code>	similar...
<code>mvcur(oldrow, oldcol, newrow, newcol)</code>	low level cursor motion
<code>mvdelch(y, x)</code>	like <i>delch</i> , but <i>move(y, x)</i> first
<code>mvgetch(y, x)</code>	etc.
<code>mvgetstr(y, x, str)</code>	
<code>mvinch(y, x)</code>	
<code>mvinsch(y, x, c)</code>	
<code>mvprintw(y, x, fmt, args)</code>	
<code>mvscanw(y, x, fmt, args)</code>	
<code>mwaddch(win, y, x, ch)</code>	
<code>mwaddstr(win, y, x, str)</code>	
<code>mwdelch(win, y, x)</code>	
<code>mwgetch(win, y, x)</code>	
<code>mwgetstr(win, y, x, str)</code>	
<code>mvwin(win, by, bx)</code>	
<code>mvwinch(win, y, x)</code>	
<code>mvwinsch(win, y, x, c)</code>	
<code>mwprintw(win, y, x, fmt, args)</code>	
<code>mwscanw(win, y, x, fmt, args)</code>	
<code>newpad(nlines, ncols)</code>	create a new pad with given dimensions
<code>newterm(type, fd)</code>	set up new terminal of given type to output on <i>fd</i>
<code>newwin(lines, cols, begin_y, begin_x)</code>	create a new window
<code>nl()</code>	set newline mapping
<code>nocbreak()</code>	unset <i>cbreak</i> mode
<code>nodelay(win, bf)</code>	enable <i>nodelay</i> input mode through <i>getch</i>

<code>noecho()</code>	unset echo mode
<code>nonl()</code>	unset newline mapping
<code>noraw()</code>	unset raw mode
<code>overlay(win1, win2)</code>	overlay win1 on win2
<code>overwrite(win1, win2)</code>	overwrite win1 on top of win2
<code>pnoutrefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)</code>	like <code>prefresh</code> but with no output until <code>doupdate</code> called
<code>prefresh(pad, pminrow, pmincol, sminrow, smincol, smaxrow, smaxcol)</code>	refresh from pad starting with given upper left corner of pad with output to given portion of screen
<code>printw(fmt, arg1, arg2, ...)</code>	printf on <i>stdscr</i>
<code>raw()</code>	set raw mode
<code>refresh()</code>	make current screen look like <i>stdscr</i>
<code>resetterm()</code>	set tty modes to "out of curses" state
<code>resetty()</code>	reset tty flags to stored value
<code>saveterm()</code>	save current modes as "in curses" state
<code>savetty()</code>	store current tty flags
<code>scanw(fmt, arg1, arg2, ...)</code>	scanf through <i>stdscr</i>
<code>scroll(win)</code>	scroll <i>win</i> one line
<code>scrollok(win, flag)</code>	allow terminal to scroll if flag != 0
<code>set_term(new)</code>	now talk to terminal new
<code>setscrreg(t, b)</code>	set user scrolling region to lines t through b
<code>setterm(type)</code>	establish terminal with given type
<code>setupterm(term, filenum, errret)</code>	
<code>standend()</code>	clear standout mode attribute
<code>standout()</code>	set standout mode attribute
<code>subwin(win, lines, cols, begin_y, begin_x)</code>	create a subwindow
<code>touchwin(win)</code>	change all of <i>win</i>
<code>traceoff()</code>	turn off debugging trace output
<code>traceon()</code>	turn on debugging trace output
<code>typeahead(fd)</code>	use file descriptor fd to check typeahead
<code>unctrl(ch)</code>	printable version of <i>ch</i>
<code>waddch(win, ch)</code>	add char to <i>win</i>
<code>waddstr(win, str)</code>	add string to <i>win</i>
<code>wattroff(win, attrs)</code>	turn off <i>attrs</i> in <i>win</i>
<code>wattron(win, attrs)</code>	turn on <i>attrs</i> in <i>win</i>
<code>wattrset(win, attrs)</code>	set <i>attrs</i> in <i>win</i> to <i>attrs</i>
<code>wclear(win)</code>	clear <i>win</i>
<code>wclrtoebot(win)</code>	clear to bottom of <i>win</i>
<code>wclrtoeol(win)</code>	clear to end of line on <i>win</i>
<code>wdelch(win, c)</code>	delete char from <i>win</i>
<code>wdeleteln(win)</code>	delete line from <i>win</i>
<code>werase(win)</code>	erase <i>win</i>
<code>wgetch(win)</code>	get a char through <i>win</i>
<code>wgetstr(win, str)</code>	get a string through <i>win</i>
<code>winch(win)</code>	get char at current (y, x) in <i>win</i>
<code>winsch(win, c)</code>	insert char into <i>win</i>
<code>winsertln(win)</code>	insert line into <i>win</i>
<code>wmove(win, y, x)</code>	set current (y, x) co-ordinates on <i>win</i>
<code>wnoutrefresh(win)</code>	refresh but no screen output
<code>wprintw(win, fmt, arg1, arg2, ...)</code>	printf on <i>win</i>
<code>wrefresh(win)</code>	make screen look like <i>win</i>
<code>wscanw(win, fmt, arg1, arg2, ...)</code>	scanf through <i>win</i>
<code>wsetscrreg(win, t, b)</code>	set scrolling region of <i>win</i>
<code>wstandend(win)</code>	clear standout attribute in <i>win</i>
<code>wstandout(win)</code>	set standout attribute in <i>win</i>

TERMINFO LEVEL ROUTINES

These routines should be called by programs wishing to deal directly with the terminfo database. Due to the low level of this interface, it is discouraged. Initially, *setupterm* should be called. This will define the set of terminal dependent variables defined in *terminfo(4)*. The include files *< curses.h>* and *< term.h>* should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through *tparm* to instantiate them. All terminfo strings (including the output of *tparm*) should be printed with *tputs* or *putp*. Before exiting, *resetterm* should be called to restore the tty modes. (Programs desiring shell escapes or suspending with control Z can call *resetterm* before the shell is called and *fixterm* after returning from the shell.)

<i>fixterm()</i>	restore tty modes for terminfo use (called by <i>setupterm</i>)
<i>resetterm()</i>	reset tty modes to state before program entry
<i>setupterm(term, fd, rc)</i>	read in database. Terminal type is the character string <i>term</i> , all output is to UNIX System file descriptor <i>fd</i> . A status value is returned in the integer pointed to by <i>rc</i> : 1 is normal. The simplest call would be <i>setupterm(0, 1, 0)</i> which uses all the defaults.
<i>tparm(str, p1, p2, ..., p9)</i>	instantiate string <i>str</i> with parms <i>p1</i> .
<i>tputs(str, affcnt, putc)</i>	apply padding info to string <i>str</i> . <i>affcnt</i> is the number of lines affected, or 1 if not applicable. <i>putc</i> is a putchar-like function to which the characters are passed, one at a time.
<i>putp(str)</i>	handy function that calls <i>tputs(str, 1, putchar)</i> .
<i>vidputs(attrs, putc)</i>	output the string to put terminal in video attribute mode <i>attrs</i> , which is any combination of the attributes listed below. Chars are passed to putchar-like function <i>putc</i> .
<i>vidattr(attrs)</i>	Like <i>vidputs</i> but outputs through <i>putc</i>

TERMCAP COMPATIBILITY ROUTINES

These routines were included as a conversion aid for programs that use *termcap*. Their parameters are the same as for *termcap*. They are emulated using the *terminfo* database. They may go away at a later date.

<i>tgetent(bp, name)</i>	look up <i>termcap</i> entry for name
<i>tgetflag(id)</i>	get boolean entry for id
<i>tgetnum(id)</i>	get numeric entry for id
<i>tgetstr(id, area)</i>	get string entry for id
<i>tgoto(cap, col, row)</i>	apply parms to given cap
<i>tputs(cap, affcnt, fn)</i>	apply padding to cap calling <i>fn</i> as <i>putc</i>

ATTRIBUTES

The following video attributes can be passed to the functions *attron*, *attroff*, *attrset*.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_BLANK	Blanking (invisible)
A_PROTECT	Protected
A_ALTCHARSET	Alternate character set

FUNCTION KEYS

The following function keys might be returned by *getch* if *keypad* has been enabled. Note that not all of these are currently supported, due to lack of definitions in *terminfo* or the terminal not transmitting a unique code when the key is pressed.

<i>Name</i>	<i>Value</i>	<i>Key name</i>
KEY_BREAK	0401	break key (unreliable)
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	backspace (unreliable)
KEY_F0	0410	Function keys. Space for 64 is reserved.
KEY_F(n)	(KEY_F0+(n))	Formula for fn.
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen
KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send (unreliable)
KEY_SRESET	0530	soft (partial) reset (unreliable)
KEY_RESET	0531	reset or hard reset (unreliable)
KEY_PRINT	0532	print or copy
KEY_LL	0533	home down or bottom (lower left)

WARNING

The plotting library *plot(3X)* and the curses library *curses(3X)* both use the names *erase()* and *move()*. The curses versions are macros. If you need both libraries, put the *plot(3X)* code in a different source file than the *curses(3X)* code, and/or *#undef move()* and *erase()* in the *plot(3X)* code.

NAME

terminfo — terminal capability data base

SYNOPSIS

/usr/lib/terminfo/*/*

DESCRIPTION

Terminfo is a data base describing terminals, used, e.g., by *vi*(1) and *curses*(3X). Terminals are described in *terminfo* by giving a set of capabilities which they have, and by describing how operations are performed. Padding requirements and initialization sequences are included in *terminfo*.

Entries in *terminfo* consist of a number of ',' separated fields. White space after each ',' is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by '|' characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, thus "hp2621". This name should not contain hyphens, except that synonyms may be chosen that do not conflict with other names. Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. Thus, a vt100 in 132 column mode would be vt100-w. The following suffixes should be used where possible:

Suffix	Meaning	Example
-w	Wide mode (more than 80 columns)	vt100-w
-am	With auto. margins (usually default)	vt100-am
-nam	Without automatic margins	vt100-nam
-n	Number of lines on the screen	aaa-60
-na	No arrow keys (leave them in local)	c100-na
-np	Number of pages of memory	c100-4p
-rv	Reverse video	c100-rv

CAPABILITIES

The variable is the name by which the programmer (at the terminfo level) accesses the capability. The capname is the short name used in the text of the database, and is used by a person updating the database. The i.code is the two letter internal code used in the compiled database, and always corresponds to the old **termcap** capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short and to allow the tabs in the source file **caps** to line up nicely. Whenever possible, names are chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

- (P) indicates that padding may be specified
- (G) indicates that the string is passed through tparm with parms as given (#).
- (*) indicates that padding may be based on the number of lines affected
- (#) indicates the parameter.

Variable Booleans	Cap- name	I. Code	Description
auto_left_margin,	bw	bw	cub1 wraps from column 0 to last column
auto_right_margin,	am	am	Terminal has automatic margins
beehive_glitch,	xs_b	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch,	xhp	xs	Standout not erased by overwriting (hp)
eat_newline_glitch,	xenl	xn	newline ignored after 80 cols (Concept)
erase_overstrike,	eo	eo	Can erase overstrikes with a blank
generic_type,	gn	gn	Generic line type (e.g., dialup, switch).
hard_copy,	hc	hc	Hardcopy terminal
has_meta_key,	km	km	Has a meta key (shift, sets parity bit)
has_status_line,	hs	hs	Has extra "status line"
insert_null_glitch,	in	in	Insert mode distinguishes nulls
memory_above,	da	da	Display may be retained above the screen
memory_below,	db	db	Display may be retained below the screen
move_insert_mode,	mir	mi	Safe to move while in insert mode
move_standout_mode,	msgr	ms	Safe to move in standout modes
over_strike,	os	os	Terminal overstrikes
status_line_esc_ok,	eslok	es	Escape can be used on the status line
teleray_glitch,	xt	xt	Tabs ruin, magic so char (Teleray 1061)
tilde_glitch,	hz	hz	Hazeltine; can not print '~'s
transparent_underline,	ul	ul	underline character overstrikes
xon_xoff,	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns,	cols	co	Number of columns in a line
init_tabs,	it	it	Tabs initially every # spaces
lines,	lines	li	Number of lines on screen or page
lines_of_memory,	lm	lm	Lines of memory if > lines. 0 means varies
magic_cookie_glitch,	xmc	sg	Number of blank chars left by smso or rmso
padding_baud_rate,	pb	pb	Lowest baud where cr/nl padding is needed
virtual_terminal,	vt	vt	Virtual terminal number (UNIX system)
width_status_line,	ws_l	ws	No. columns in status line
Strings:			
back_tab,	cbt	bt	Back tab (P)
bell,	bel	bl	Audible signal (bell) (P)
carriage_return,	cr	cr	Carriage return (P*)
change_scroll_region,	csr	cs	change to lines #1 through #2 (vt100) (PG)
clear_all_tabs,	tbc	ct	Clear all tab stops (P)
clear_screen,	clear	cl	Clear screen and home cursor (P*)
clr_eol,	el	ce	Clear to end of line (P)
clr_eos,	ed	cd	Clear to end of display (P*)
column_address,	hpa	ch	Set cursor column (PG)
command_character,	cmdch	CC	Term. settable cmd char in prototype
cursor_address,	cup	cm	Screen rel. cursor motion row #1 col #2 (PG)
cursor_down,	cudl	do	Down one line
cursor_home,	home	ho	Home cursor (if no cup)
cursor_invisible,	civis	vi	Make cursor invisible
cursor_left,	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal,	cnorm	ve	Make cursor appear normal (undo vs/vi)
cursor_right,	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll,	ll	ll	Last line, first column (if no cup)
cursor_up,	cuu1	up	Upline (cursor up)
cursor_visible,	cvvis	vs	Make cursor very visible
delete_character,	dch1	dc	Delete character (P*)

delete_line,	dll	dl	Delete line (P*)
dis_status_line,	dsl	ds	Disable status line
down_half_line,	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode,	smacs	as	Start alternate character set (P)
enter_blink_mode,	blink	mb	Turn on blinking
enter_bold_mode,	bold	md	Turn on bold (extra bright) mode
enter_ca_mode,	smcup	ti	String to begin programs that use cup
enter_delete_mode,	smdc	dm	Delete mode (enter)
enter_dim_mode,	dim	mh	Turn on half-bright mode
enter_insert_mode,	smir	im	Insert mode (enter);
enter_protected_mode,	prot	mp	Turn on protected mode
enter_reverse_mode,	rev	mr	Turn on reverse video mode
enter_secure_mode,	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode,	sms0	so	Begin stand out mode
enter_underline_mode,	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (PG)
exit_alt_charset_mode,	rmacs	ae	End alternate character set (P)
exit_attribute_mode,	sgr0	me	Turn off all attributes
exit_ca_mode,	rmcup	te	String to end programs that use cup
exit_delete_mode,	rmdc	ed	End delete mode
exit_insert_mode,	rmir	ei	End insert mode
exit_standout_mode,	rmso	se	End stand out mode
exit_underline_mode,	rmul	ue	End underscore mode
flash_screen,	flash	vb	Visible bell (may not move cursor)
form_feed,	ff	ff	Hardcopy terminal page eject (P*)
from_status_line,	fsl	fs	Return from status line
init_1string,	is1	i1	Terminal initialization string
init_2string,	is2	i2	Terminal initialization string
init_3string,	is3	i3	Terminal initialization string
init_file,	if	if	Name of file containing is
insert_character,	ich1	ic	Insert character (P)
insert_line,	ill	al	Add new blank line (P*)
insert_padding,	ip	ip	Insert pad after character inserted (P*)
key_backspace,	kbs	kb	Sent by backspace key
key_catab,	ktbc	ka	Sent by clear-all-tabs key
key_clear,	kclr	kC	Sent by clear screen or erase key
key_ctab,	kctab	kt	Sent by clear-tab key
key_dc,	kdch1	kD	Sent by delete character key
key_dl,	kdl1	kL	Sent by delete line key
key_down,	kcud1	kd	Sent by terminal down arrow key
key_eic,	krmir	kM	Sent by rmir or smir in insert mode
key_eol,	kel	kE	Sent by clear-to-end-of-line key
key_eos,	ked	kS	Sent by clear-to-end-of-screen key
key_f0,	kf0	k0	Sent by function key f0
key_f1,	kf1	k1	Sent by function key f1
key_f10,	kf10	ka	Sent by function key f10
key_f2,	kf2	k2	Sent by function key f2
key_f3,	kf3	k3	Sent by function key f3
key_f4,	kf4	k4	Sent by function key f4
key_f5,	kf5	k5	Sent by function key f5
key_f6,	kf6	k6	Sent by function key f6
key_f7,	kf7	k7	Sent by function key f7
key_f8,	kf8	k8	Sent by function key f8
key_f9,	kf9	k9	Sent by function key f9
key_home,	khome	kh	Sent by home key
key_ic,	kich1	kI	Sent by ins char/enter ins mode key
key_il,	kill	kA	Sent by insert line

key_left,	kcub1	kl	Sent by terminal left arrow key
key_ll,	kll	kH	Sent by home-down key
key_npage,	knp	kN	Sent by next-page key
key_ppage,	kpp	kP	Sent by previous-page key
key_right,	kcufl	kr	Sent by terminal right arrow key
key_sf,	kind	kF	Sent by scroll-forward/down key
key_sr,	kri	kR	Sent by scroll-backward/up key
key_stab,	khts	kT	Sent by set-tab key
key_up,	kcuul	ku	Sent by terminal up arrow key
keypad_local,	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit,	smkx	ks	Put terminal in "keypad transmit" mode
lab_f0,	lf0	l0	Labels on function key f0 if not f0
lab_f1,	lf1	l1	Labels on function key f1 if not f1
lab_f10,	lf10	la	Labels on function key f10 if not f10
lab_f2,	lf2	l2	Labels on function key f2 if not f2
lab_f3,	lf3	l3	Labels on function key f3 if not f3
lab_f4,	lf4	l4	Labels on function key f4 if not f4
lab_f5,	lf5	l5	Labels on function key f5 if not f5
lab_f6,	lf6	l6	Labels on function key f6 if not f6
lab_f7,	lf7	l7	Labels on function key f7 if not f7
lab_f8,	lf8	l8	Labels on function key f8 if not f8
lab_f9,	lf9	l9	Labels on function key f9 if not f9
meta_on,	smm	mm	Turn on "meta mode" (8th bit)
meta_off,	rmm	mo	Turn off "meta mode"
newline,	nel	nw	Newline (behaves like cr followed by lf)
pad_char,	pad	pc	Pad character (rather than null)
parm_dch,	dch	DC	Delete #1 chars (PG*)
parm_delete_line,	dl	DL	Delete #1 lines (PG*)
parm_down_cursor,	cud	DO	Move cursor down #1 lines (PG*)
parm_ich,	ich	IC	Insert #1 blank chars (PG*)
parm_index,	indn	SF	Scroll forward #1 lines (PG)
parm_insert_line,	il	AL	Add #1 new blank lines (PG*)
parm_left_cursor,	cub	LE	Move cursor left #1 spaces (PG)
parm_right_cursor,	cuf	RI	Move cursor right #1 spaces (PG*)
parm_rindex,	rin	SR	Scroll backward #1 lines (PG)
parm_up_cursor,	cuu	UP	Move cursor up #1 lines (PG*)
pkey_key,	pfkey	pk	Prog funct key #1 to type string #2
pkey_local,	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit,	pfx	px	Prog funct key #1 to xmit string #2
print_screen,	mc0	ps	Print contents of the screen
prtr_off,	mc4	pf	Turn off the printer
prtr_on,	mc5	po	Turn on the printer
repeat_char,	rep	rp	Repeat char #1 #2 times. (PG*)
reset_lstring,	rs1	r1	Reset terminal completely to sane modes.
reset_2string,	rs2	r2	Reset terminal completely to sane modes.
reset_3string,	rs3	r3	Reset terminal completely to sane modes.
reset_file,	rf	rf	Name of file containing reset string
restore_cursor,	rc	rc	Restore cursor to position of last sc
row_address,	vpa	cv	Vertical position absolute (set row) (PG)
save_cursor,	sc	sc	Save cursor position (P)
scroll_forward,	ind	sf	Scroll text up (P)
scroll_reverse,	ri	sr	Scroll text down (P)
set_attributes,	sgr	sa	Define the video attributes (PG9)
set_tab,	hts	st	Set a tab in all rows, current column
set_window,	wind	wi	Current window is lines #1-#2 cols #3-#4
tab,	ht	ta	Tab to next 8 space hardware tab stop
to_status_line,	tsl	ts	Go to status line, column #1

underline_char,	uc	uc	Underscore one char and move past it
up_half_line,	hu	hu	Half-line up (reverse 1/2 linefeed)
init_prog,	iprog	iP	Path name of program for init
key_a1,	ka1	K1	Upper left of keypad
key_a3,	ka3	K3	Upper right of keypad
key_b2,	kb2	K2	Center of keypad
key_c1,	kc1	K4	Lower left of keypad
key_c3,	kc3	K5	Lower right of keypad
prtr_non,	mc5p	pO	Turn on the printer for #1 bytes

A Sample Entry

The following entry, which describes the Concept-100, is among the more complex entries in the *terminfo* file as of this writing.

```
concept 100 | c 100 | concept | c 104 | c 100-4p | concept 100,
  am, bel=^G, blank=^EH, blink=^EC, clear=^L$<2*>, cnorm=^Ew,
  cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=^E=,
  cup=^Ea%p1% ' '%+%c%p2% ' '%+%c,
  cuu1=^E; , cvvis=^EW, db, dch1=^E^A$<16*>, dim=^EE, dl1=^E^B$<3*>,
  ed=^E^C$<16*>, el=^E^U$<16>, eo, flash=^Ek$<20>^EK, ht=^t$<8>,
  il1=^E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
  is2=^EU^Ef^E7^E5^E8^E1^ENH^EK^E200^Eo&^200^Eo47^E,
  kbs=^h, kcub1=^E>, kcud1=^E<, kcuf1=^E=, kcuu1=^E; ,
  kf1=^E5, kf2=^E6, kf3=^E7, khome=^E7,
  lines#24, mir, pb#9600, prot=^EI, rep=^Er%p1%c%p2% ' '%+%c$<.2*>,
  rev=^ED, rmcup=^Ev $<6>^Ep^r^n, rmir=^E200, rmkx=^Ex,
  rmso=^Ed^Ee, rmul=^Eg, rmul=^Eg, sgr0=^EN200,
  smcup=^EU^Ev 8p^Ep^r, smir=^E^P, smkx=^EX, smso=^EE^ED,
  smul=^EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Comments may be included on lines beginning with "#". Capabilities in *terminfo* are of three types: Boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular delays, and string capabilities, which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the capname code, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in '\$<..>' brackets, as in **el=^EK\$<3>**, and padding characters are supplied by *tputs* to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit to tenths of milliseconds. (Only one decimal place is allowed.)

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both `\E` and `\e` map to an ESCAPE character, `^x` maps to a control-x for any appropriate x, and the sequences `\n` `\l` `\r` `\t` `\b` `\f` `\s` give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include `\^` for `^`, `\\` for `\`, `\,` for comma, `\:` for `:`, and `\0` for null. (`\0` will produce `\200`, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a `\`.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second `ind` in the example above.

Preparing Descriptions

We now outline how to prepare descriptions of terminals. The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with *vi* to check that they are correct. Be aware that a very unusual terminal may expose deficiencies in the ability of the *terminfo* file to describe it or bugs in *vi*. To easily test a new terminal description you can set the environment variable `TERMINFO` to a pathname of a directory containing the compiled description you are working on and programs will look there rather than in */usr/lib/terminfo*. To get the padding for insert line right (if the terminal manufacturer did not document it) a severe test is to edit */etc/passwd* at 9600 baud, delete 16 or so lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

Basic Capabilities

The number of columns on each line for the terminal is given by the `cols` numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the `lines` capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the `am` capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the `clear` string capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the `os` capability. If the terminal is a printing terminal, with no soft copy unit, give it both `hc` and `os`. (`os` applies to storage scope terminals, such as TEKTRONIX 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as `cr`. (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as `bel`.

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as `cub1`. Similarly, codes to move to the right, up, and down should be given as `cuf1`, `cuu1`, and `cud1`. These local cursor motions should not alter the text they pass over, for example, you would not normally use `'cuf1= '` because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless `bw` is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the `ind` (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the `ri` (reverse index) string. The strings `ind` and `ri` are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and glass-tty terminals. Thus the model 33 teletype is described as

```
33|tty33|tty|model 33 teletype,
```

```
    bel=^G, cols#72, cr=^M, cud1=^J, hc, ind=^J, os,
```

while the Lear Siegler ADM-3 is described as

```
adm3|3|lsi adm3,
```

```
    am, bel=^G, clear=^Z, cols#80, cr=^M, cub1=^H, cud1=^J,
    ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability, with *printf*(3S) like escapes **%x** in it. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup**.

The parameter mechanism uses a stack and special **%** codes to manipulate it. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary.

The **%** encodings have the following meanings:

%%	outputs '%'
%d	print pop() as in printf
%2d	print pop() like %2d
%3d	print pop() like %3d
%02d	
%03d	as in printf
%c	print pop() gives %c
%s	print pop() gives %s
%p[1-9]	push ith parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	char constant c
%{nn}	integer constant nn
%+ %- %* %/ %m	arithmetic (%m is mod): push(pop() op pop())

%& % %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %~	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)

%? expr %t thenpart %e elsepart %;

if-then-else, %e elsepart is optional.

else-if's are possible ala Algol 68:

%? c₁ %t b₁ %e c₂ %t b₂ %e c₃ %t b₃ %e c₄ %t b₄ %e %;

c_i are conditions, b_i are bodies.

Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is cup=\E&%p2%2dc%p1%2dY\$<6>.

The Microterm ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, cup=^T%p1%c%p2%c. Terminals which use %c need to be able to backspace the cursor (**cu**1), and to move the cursor up one line on the screen (**cuu**1). This is necessary because it is not always safe to transmit \n ^D and \r, as the system may change or discard them. (The library routines dealing with terminfo set tty modes so that tabs are never expanded, so \t is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus cup=\E=%p1%' '%+%c%p2%' '%+%c. After sending \E=, this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the hp2645) and can be used in preference to **cup**. If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud**, **cub**, **cuf**, and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup**, such as the TEKTRONIX 4025.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu**1 from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the \EH sequence on Hewlett-Packard terminals cannot be used for **home**.)

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el**. If the terminal can clear from the current position to the end of the display, then this should be given as **ed**. **Ed** is only defined from the first column of a line. (Thus, it can be

simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/delete line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl**. If the terminal has a settable scrolling region (like the vt100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind**. The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo*. The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. You can determine the kind of terminal you have by clearing the screen and then typing text separated by cursor motions. Type **abc def** using local cursor motions (not spaces) between the **abc** and the **def**. Then position the cursor before the **abc** and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then your terminal does not distinguish between blanks and untyped positions. If the **abc** shifts over to the **def** which then move together around the end of the current line and onto the next as you insert, you have the second type of terminal, and should give the capability **in**, which stands for insert null. While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

Terminfo can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If your terminal has both, insert mode is usually preferable to **ich1**. Do not give both unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of

milliseconds in **ip** (a string option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If your terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If your terminal allows motion while in insert mode you can give the capability **mir** to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, you can specify **dch1** to delete a single character, **dch** with one parameter, *n*, to delete *n* characters, and delete mode by giving **smdc** and **rmde** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If your terminal has one or more kinds of display attributes, these can be represented in a number of different ways. You should choose one display form as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If you have a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmsso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TVI 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul** respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Microterm Mime, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking) **bold** (bold or extra bright) **dim** (dim or half-bright) **invis** (blanking or invisible text) **prot** (protected) **rev** (reverse video) **sgr0** (turn off *all* attribute modes) **smaes** (enter alternate character set mode) and **rmaes** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking 9 parameters. Each parameter is either 0 or 1, as the corresponding attribute is on or off. The 9 parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis**. If there is a way to make the cursor completely invisible, give that as **civis**. The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup**. This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the TEKTRONIX 4025, where **smcup** sets the command character to be the one used by terminfo.

If your terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then you should give the capability **ul**. If overstrikes are erasable with a blank, then this should be indicated by giving **eo**.

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx**. Otherwise the keypad is assumed to always transmit. The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1**, **kcuf1**, **kcuu1**, **kcud1**, and **khome** respectively. If there are function keys such as **f0**, **f1**, ..., **f10**, the codes they send can be given as **kf0**, **kf1**, ..., **kf10**. If these keys have labels other than the default **f0** through **f10**, the labels can be given as **lf0**, **lf1**, ..., **lf10**. The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kill** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1**, **ka3**, **kb2**, **kc1**, and **kc3**. These keys are useful when the effects of a 3 by 3 directional pad are needed.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt**. By convention, if the teletype modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter it is given, showing the number of spaces the tabs are set to. This is normally used by the *tset* command to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the terminfo description can assume that they are properly set.

Other capabilities include **is1**, **is2**, and **is3**, initialization strings for the terminal, **iprog**, the path name of a program to be run to initialize the terminal, and **if**, the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the terminfo description. They are normally sent to the terminal, by the *tset* program, each time the user logs in. They will be printed in the following order: **is1**; **is2**; setting tabs using **tbc** and **hts**; **if**; running the program **iprog**; and finally **is3**. Most initialization is done with **is2**. Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3**. A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1**, **rs2**, **rf**, and **rs3**, analogous to **is2** and **if**. These strings are output by the *reset* program, which is used when the terminal gets into a wedged state. Commands are normally placed in **rs2** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set the vt100 into 80-column mode would normally be part of **is2**, but it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode.

If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row). If a more complex sequence is needed to set the tabs than can be described by this, the sequence can be placed in **is2** or **if**.

Delays

Certain capabilities control padding in the teletype driver. These are primarily needed by hard copy terminals, and are used by the *tset* program to set teletype modes appropriately. Delays embedded in the capabilities **cr**, **ind**, **cub1**, **ff**, and **tab** will cause the appropriate delay bits to be set in the teletype driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb**.

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad**. Only the first character of the **pad** string is used.

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated by giving **hs**. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl**. (**fsl** must leave the cursor position in the same place it was before **tsl**. If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as **tab**, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl**. If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc**. The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols**. If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl**.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be

indicated with the parameterized string **rep**. The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, `tparam(repeat_char, 'x', 10)` is the same as `'xxxxxxxxxx'`.

If the terminal has a settable command character, such as the TEKTRONIX 4025, this can be indicated with **cmdch**. A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it. The following convention is supported on some UNIX systems: The environment is to be searched for a **CC** variable, and if found, all occurrences of the prototype character are replaced with the character in the environment variable.

Terminal descriptions that do not represent a specific kind of known terminal, such as *switch*, *dialup*, *patch*, and *network*, should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal. (This capability does not apply to *virtual* terminal descriptions for which the escape sequences are known.)

If the terminal uses *xon/xoff* handshaking for flow control, give **xon**. Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has a "meta key" which acts as a shift key, setting the 8th bit of any character transmitted, this fact can be indicated with **km**. Otherwise, software will assume that the 8th bit is parity and it will usually be cleared. If strings exist to turn this "meta mode" on and off, they can be given as **smm** and **rmm**.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm**. A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

If the terminal is one of those supported by the UNIX system virtual terminal protocol, the terminal number can be given as **vt**.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4**, is transparently passed to the printer while an **mc5p** is in effect.

Strings to program function keys can be given as **pfkey**, **pfloc**, and **pfx**. Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Glitches and Braindamage

Hazeltine terminals, which do not allow ‘” characters to be displayed should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept and vt100, should indicate **xenl**.

If **el** is required to get rid of standout (instead of merely writing normal text on top of it), **xhp** should be given.

Teleray terminals, where tabs turn all characters moved over to blanks, should indicate **xt** (destructive tabs). This glitch is also taken to mean that it is not possible to position the cursor on top of a “magic cookie”, that to erase stand-out mode it is instead necessary to use delete and insert line.

The Beehive Superbee, which is unable to correctly transmit the escape or control C characters, has **xsib**, indicating that the f1 key is used for escape and f2 for control C. (Only certain Superbees have this problem, depending on the ROM.)

Other specific terminal problems may be corrected by adding more capabilities of the form **xx**.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition, where **xx** is the capability. For example, the entry

2621-nl, smkx@, rmkx@, use=2621,

defines a 2621-nl that does not have the **smkx** or **rmkx** capabilities, and hence does not turn on the function key labels when in visual mode. This is useful for different modes for a terminal, or for different user preferences.

FILES

/usr/lib/terminfo/?.* files containing terminal descriptions

SEE ALSO

curses(3X), printf(3S), term(5).

3B2 Computer System Terminal Information Utilities Guide.

NAME

tic — terminfo compiler

SYNOPSIS

tic [*-v*[*n*]] file ...

DESCRIPTION

Tic translates terminfo files from the source format into the compiled format. The results are placed in the directory */usr/lib/terminfo*.

The *-v* (verbose) option causes *tic* to output trace information showing its progress. If the optional integer is appended, the level of verbosity can be increased.

Tic compiles all terminfo descriptions in the given files. When a *use=* field is discovered, *tic* searches first the current file, then the master file, which is *"./terminfo.src"*.

If the environment variable *TERMINFO* is set, the results are placed there instead of */usr/lib/terminfo*.

Some limitations: total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

/usr/lib/terminfo//** compiled terminal capability data base

SEE ALSO

curses(3X), *terminfo(4)*.

BUGS

Instead of searching *./terminfo.src*, it should check for an existing compiled entry.

NAME

tput — query terminfo database

SYNOPSIS

tput [**-Ttype**] **capname**

DESCRIPTION

Tput uses the *terminfo* database to make terminal-dependent capabilities and information available to the shell. *Tput* outputs a string if the attribute (**capability name**) is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, *tput* simply sets the exit code (0 for TRUE, 1 for FALSE), and does no output.

-Ttype indicates the type of terminal. Normally this flag is unnecessary, as the default is taken from the environment variable **\$TERM**.

Capname indicates the attribute from the *terminfo* database. See *3B2 Computer System Terminal Information Guide*.

EXAMPLES

tput clear	Echo clear-screen sequence for the current terminal.
tput cols	Print the number of columns for the current terminal.
tput -T450 cols	Print the number of columns for the 450 terminal.
bold='tput smso'	Set shell variable "bold" to stand-out mode sequence for current terminal. This might be followed by a prompt: echo "\${bold}Please type in your name: \c"
tput hc	Set exit code to indicate if current terminal is a hardcopy terminal.

FILES

/etc/term/?/*	Terminal descriptor files
/usr/include/term.h	Definition files
/usr/include/curses.h	

SEE ALSO

stty(1).
3B2 Computer System Terminal Information Utilities Guide.
3B2 Computer System Programmer Reference Manual.

DIAGNOSTICS

Tput prints error messages and returns the following error codes on error:

-1	Usage error.
-2	Bad terminal type.
-3	Bad capname.

In addition, if a numeric capname is requested for a terminal that has no value for that capname (e.g., **tput -T450 lines**), **-1** is printed. However, no error codes are returned for string capnames with no values.

Appendix B

CURSES EXAMPLES

	PAGE
EXAMPLE PROGRAM 'editor'	B-2
EXAMPLE PROGRAM 'highlight'	B-10
EXAMPLE PROGRAM 'scatter'	B-12
EXAMPLE PROGRAM 'show'	B-14
EXAMPLE PROGRAM 'termhl'	B-16
EXAMPLE PROGRAM 'two'	B-19
EXAMPLE PROGRAM 'window'	B-22

Appendix B

CURSES EXAMPLES

This appendix contains some examples of **curses** programs. Although these examples are functional programs, they are not complete enough to be considered useful. These examples are intended for demonstration purposes only. However, these examples may be used by a skillful programmer as a base to create a useful **curses** program.

The examples contain comments which explain the intended function of a particular step in the program. The comments are for clarity only and are not a functional piece of the program.

EXAMPLE PROGRAM 'editor'

This program is a very simple screen editor patterned after the **vi** editor. The program illustrates how to use **curses** to write a screen editor. This editor keeps the buffer in **stdscr** to keep the program simple — obviously, a real screen editor would keep a separate data structure. Many simplifications have been made here — no provision is made for files of any length other than the size of the screen, for lines longer than the width of the screen, or for control characters in the file.

Several points about this program are worth making. The routine to write out the file illustrates the use of the **mvinch** function which returns the character in a window at a given position. The data structure used here does not have a provision for keeping track of the number of characters in a line or the number of lines in the file; so, trailing blanks are eliminated when the file is written out.

The program uses built-in **curses** functions **insch**, **delch**, **insertln**, and **deleteln**. These functions behave like similar functions on intelligent terminals, inserting and deleting a character or line.

The command interpreter accepts not only ASCII characters but also special keys. This is important — a good program will accept both. (Some editors are “modeless,” using nonprinting characters for commands. This is largely a matter of taste — the point being made here is that both arrow keys and ordinary ASCII characters should be handled.) It is important to know how to handle special keys. Special keys make it easier for someone else to learn to use your program if they can use the arrow keys instead of having to memorize that “h” means left, “j” means down, “k” means up, and “l” means right. On the other hand, not all terminals have arrow keys; so, your program will be usable on a larger class of terminals if there is an ASCII character which is a synonym for each special key. Also, experienced users dislike having to move their hands from the “home row” position to use special keys, since they can work faster with alphabetic keys.

Note the call to **mvaddstr** in the input routine. The **addstr** function is roughly like the C Language **fputs** function which writes out a string of characters. Like **fputs**, **addstr** does not add a trailing newline. It

is the same as a series of calls to **addch** using the characters in the string. (Refer to the **mvaddstr** function description in the "INPUT/OUTPUT" section of Chapter 2.)

The control-L command illustrates a feature that most programs using **curses** should add. Often, some program beyond the control of **curses** has written something to the screen, or some line noise has messed up the screen beyond the tracking capability of **curses**. In this case, the user usually types control-L, causing the screen to be cleared and redrawn. This is done with the call to **clearok(curscr)**, which sets a flag causing the next **refresh** to first clear the screen. Then **refresh** is called to force the redraw.

Note also the call to **flash()**, which flashes the screen if possible, and otherwise rings the bell. Flashing the screen is intended as a bell replacement and is particularly useful if the bell bothers someone within earshot of the user. The routine **beep()** can be called when a real beep is desired. (If for some reason the terminal is unable to beep, but able to flash, a call to **beep** will flash the screen.)

Another important point is that the input command is terminated by control-D, not escape. It is very tempting to use escape as a command, since escape is one of the few special keys which is available on every keyboard. (Return and break are the only others.) However, using escape as a separate key introduces an ambiguity. Most terminals use sequences of characters beginning with escape ("escape sequences") to control the terminal and have special keys that send escape sequences to the computer. If the computer sees an escape coming from the terminal, it cannot tell for sure whether the user pushed the escape key or whether a special key was pressed. Curses handles the ambiguity by waiting for up to one second. If another character is received during this second, and if that character might be the beginning of a special key, more input is read (waiting for up to one second for each character) until either a full special key is read, one second passes, or a character is received that could not have been generated by a special key. While this strategy works most of the time, it is not foolproof. It is possible for the user to press escape and then to type another key quickly, which causes **curses** to think a special key has been pressed. Also, there is a one second pause until the escape can be passed to the user program, resulting in slower

response to the escape key. Many existing programs use escape as a fundamental command which cannot be changed without infuriating a large class of users. Such programs cannot make use of special keys without dealing with this ambiguity, and at best these programs must resort to a timeout solution. The moral is clear: when designing your program, avoid the escape key.

The example program for the simple editor is shown on the following pages.

```
/*
 * editor: A screen-oriented editor. The user
 * interface is similar to a subset of vi.
 * The buffer is kept in stdscr itself to simplify
 * the program.
 */

#include <curses.h>

#define CTRL(c) ('c' & 037)

main(argc, argv)
char **argv;
{
    int i, n, l;
    int c;
    FILE *fd;

    if (argc != 2) {
        fprintf(stderr, " Usage: edit file\n" );
        exit(1);
    }

    fd = fopen(argv[1], " r" );
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    cbreak();
    nonl();
    noecho();
    idlok(stdscr, TRUE);
    keypad(stdscr, TRUE);

    /* Read in the file */
    while ((c = getc(fd)) != EOF)
        addch(c);
    fclose(fd);

    move(0,0);
```

```
    refresh();
    edit();

    /* Write out the file */
    fd = fopen(argv[1], "w");
    for (l=0; l<23; l++) {
        n = len(l);
        for (i=0; i<n; i++)
            putc(mvinch(l, i), fd);
        putc('\n', fd);
    }
    fclose(fd);

    endwin();
    exit(0);
}

len(lineno)
int lineno;
{
    int linelen = COLS-1;

    while (linelen >=0 && mvinch(lineno, linelen) == ' ')
        linelen--;
    return linelen + 1;
}

/* Global value of current cursor position */
int row, col;

edit()
{
    int c;

    for (;;) {
        move(row, col);
        refresh();
        c = getch();
        switch (c) { /* Editor commands */

            /* hjkl and arrow keys: move cursor */
            /* in direction indicated */
```

```
case 'h':
case KEY_LEFT:
    if (col > 0)
        col--;
    break;

case 'j':
case KEY_DOWN:
    if (row < LINES-1)
        row++;
    break;

case 'k':
case KEY_UP:
    if (row > 0)
        row--;
    break;

case 'l':
case KEY_RIGHT:
    if (col < COLS-1)
        col++;
    break;

/* i: enter input mode */
case KEY_IC:
case 'i':
    input();
    break;

/* x: delete current character */
case KEY_DC:
case 'x':
    delch();
    break;

/* o: open up a new line and enter input mode */
case KEY_IL:
case 'o':
    move(++row, col=0);
    insertln();
    input();
```

```
        break;

/* d: delete current line */
case KEY_DL:
case 'd':
    deleteln();
    break;

/* ^L: redraw screen */
case KEY_CLEAR:
case CTRL(L):
    clearok(curscr);
    refresh();
    break;

/* w: write and quit */
case 'w':
    return;

/* q: quit without writing */
case 'q':
    endwin();
    exit(1);
default:
    flash();
    break;
    }
}

/*
 * Insert mode: accept characters and insert them.
 * End with ^D or EIC
 */
input()
{
    int c;

    stdout();
    mvaddstr(LINES-1, COLS-20, " INPUT MODE");
    standend();
    move(row, col);
```



```
refresh();
for (;;) {
    c = getch();
    if (c == CTRL(D) || c == KEY_EIC)
        break;
    insch(c);
    move(row, ++col);
    refresh();
}
move(LINES-1, COLS-20);
clrtoeol();
move(row, col);
refresh();
}
```

EXAMPLE PROGRAM 'highlight'

This program takes a text file as input and allows embedded escape sequences to control attributes. In this example program, **\U** turns on underlining, **\B** turns on bold, and **\N** restores normal text. Note the initial call to **scrollok**. This allows the terminal to scroll if the file is longer than one screen. When an attempt is made to draw past the bottom of the screen, **curses** will automatically scroll the terminal up a line and call **refresh**.

```
/*
 * highlight: a program to turn U, B, and
 * N sequences into highlighted
 * output, allowing words to be
 * displayed underlined or in bold.
 */
#include <curses.h>

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;

    if (argc != 2) {
        fprintf(stderr, " Usage: highlight file\n" );
        exit(1);
    }

    fd = fopen(argv[1], " r" );
    if (fd == NULL) {
        perror(argv[1]);
        exit(2);
    }

    initscr();
    scrollok(stdscr, TRUE);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
    }
}
```

```
    if (c == '\\') {
        c2 = getc(fd);
        switch (c2) {
            case 'B':
                attrset(A_BOLD);
                continue;
            case 'U':
                attrset(A_UNDERLINE);
                continue;
            case 'N':
                attrset(0);
                continue;
        }
        addch(c);
        addch(c2);
    }
    else
        addch(c);
}
fclose(fd);
refresh();
endwin();
exit(0);
}
```

EXAMPLE PROGRAM 'scatter'

This program reads a file, and displays the file in a random order on the screen. Some programs assume all screens are 24 lines by 80 columns. It is important to understand that many are not. The variables **LINES** and **COLS** are defined by **initscr** with the current screen size. Programs should use them instead of assuming a 24x80 screen.

```
/*
 * SCATTER. This program takes the first
 * 23 lines from the standard
 * input and displays them on the
 * VDU screen, in a random manner.
 */

#include <curses.h>

#define MAXLINES 120
#define MAXCOLS 160
char s[MAXLINES][MAXCOLS]; /* Screen Array */

main()
{
    register int row=0,col=0;
    register char c;
    int char_count=0;
    long t;
    char buf[BUFSIZ];

    initscr();
    for(row=0;row<MAXLINES;row++)
        for(col=0;col<MAXCOLS;col++)
            s[row][col]=' ';

    row = 0;
    /* Read screen in */
    while( (c=getchar()) != EOF && row < LINES ) {
        if(c != '\n') {
            /* Place char in screen array */
            s[row][col++] = c;
            if(c != ' ')

```

```
        char_count++;
    } else {
        col=0;
        row++;
    }
}

time(&t); /* Seed the random number generator */
srand((int)(t&0177777L));

while(char_count) {
    row=rand() % LINES;
    col=(rand()>>2) % COLS;
    if(s[row][col] != ' ')
    {
        move(row, col);
        addch(s[row][col]);
        s[row][col]=EOF;
        char_count--;
        refresh();
    }
}
endwin();
exit(0);
}
```

EXAMPLE PROGRAM 'show'

The **show** program pages through a file, showing one full screen each time the user presses the space bar. By creating an input file for **show** made up of 24-line pages, each segment varying slightly from the previous page, nearly any exercise for **curses** can be created. Such input files are called "show scripts."

In this program, **cbreak** is called so that the user can press the space bar without having to hit return. The **noecho** function is called to prevent the space from echoing in the middle of a **refresh**, messing up the screen. The **nonl** function is called to enable more screen optimization. The **idlok** function is called to allow insert and delete line, since many show scripts are constructed to duplicate bugs caused by that feature. The **clrtoeol** and **clrtobot** functions clear from the cursor to the end of the line and screen, respectively.

```
#include <curses.h>
#include <signal.h>

main(argc, argv)
int argc;
char *argv[];
{
    FILE *fd;
    char linebuf[BUFSIZ];
    int line;
    void done(), perror(), exit();

    if(argc != 2)
    {
        fprintf(stderr, "usage: %s file\n", argv[0]);
        exit(1);
    }
    if((fd=fopen(argv[1], "r")) == NULL)
    {
        perror(argv[1]);
        exit(2);
    }
    signal(SIGINT, done);
```

```
    initscr();
    noecho();
    cbreak();
    nonl();
    idlok(stdscr, TRUE);

    while(1)
    {
        move(0,0);
        for(line=0; line<LINES; line++)
        {
            if(fgets(linebuf, sizeof linebuf, fd) == NULL)
            {
                clrtoebot();
                done();
            }
            move(line, 0);
            printw(" %s", linebuf);
        }
        refresh();
        if(getch() == 'q')
            done();
    }
}

void
done()
{
    move(LINES-1, 0);
    clrtoeol();
    refresh();
    endwin();
    exit(0);
}
```

EXAMPLE PROGRAM 'termhl'

```
/*
 * A terminfo level version of highlight.
 */
#include <curses.h>
#include <term.h>

int ulmode = 0;          /* Currently underlining */

main(argc, argv)
char **argv;
{
    FILE *fd;
    int c, c2;
    int outch();

    if (argc > 2) {
        fprintf(stderr, " Usage: termhl [file]\n");
        exit(1);
    }

    if (argc == 2) {
        fd = fopen(argv[1], "r");
        if (fd == NULL) {
            perror(argv[1]);
            exit(2);
        }
    } else {
        fd = stdin;
    }

    setupterm(0, 1, 0);

    for (;;) {
        c = getc(fd);
        if (c == EOF)
            break;
        if (c == '\\') {
            c2 = getc(fd);
            switch (c2) {
                case 'B':
                    tputs(enter_bold_mode, 1, outch);
            }
        }
    }
}
```



```
        continue;
    case 'U':
        tputs(enter_underline_mode, 1, outch);
        ulmode = 1;
        continue;
    case 'N':
        tputs(exit_attribute_mode, 1, outch);
        ulmode = 0;
        continue;
    }
    putch(c);
    putch(c2);
}
else
    putch(c);
}
fclose(fd);
fflush(stdout);
resetterm();
exit(0);
}

/*
 * This function is like putchar, but it checks for underlining.
 */
putch(c)
int c;
{
    outch(c);
    if (ulmode && underline_char) {
        outch('\b');
        tputs(underline_char, 1, outch);
    }
}
```

```
/*  
 * Outchar is a function version of putchar that can be passed to  
 * tputs as a routine to call.  
 */  
outch(c)  
int c;  
{  
    putchar(c);  
}
```

EXAMPLE PROGRAM 'two'

This program pages through a file, showing one page to the first terminal and the next page to the second terminal. It then waits for a space to be typed on either terminal, and shows the next page to the terminal typing the space. Each terminal has to be separately put into nodelay mode. Since no standard multiplexor is available in current versions of the UNIX System, it is necessary to either busy wait or call **sleep(1)**; between each check for keyboard input. This program sleeps for a second between checks.

```
#include <curses.h>
#include <signal.h>

struct screen *me, *you;
struct screen *set_term();

FILE *fd, *fdyou;
char linebuf[512];

main(argc, argv)
char **argv;
{
    int done();
    int c;

    if (argc != 4) {
        fprintf(stderr, " Usage: two othertty otherttytype inputfile\n");
        exit(1);
    }

    fd = fopen(argv[3], "r");
    fdyou = fopen(argv[1], "w+");
    signal(SIGINT, done); /* die gracefully */

    me = newterm(getenv("TERM"), stdout); /* initialize my tty */
    you = newterm(argv[2], fdyou); /* Initialize his terminal */

    set_term(me);          /* Set modes for my terminal */
    noecho();              /* turn off tty echo */
    cbreak();              /* enter cbreak mode */
    nonl();                /* Allow linefeed */
```

```

    nodelay(stdscr,TRUE); /* No hang on input */

    set_term(you);        /* Set modes for other terminal */
    noecho();
    cbreak();
    nonl();
    nodelay(stdscr,TRUE);

    /* Dump first screen full on my terminal */
    dump_page(me);

    /* Dump second screen full on his terminal */
    dump_page(you);

    for (;;) {            /* for each screen full */
        set_term(me);
        c = getch();
        if (c == 'q')      /* wait for user to read it */
            done();
        if (c == ' ')
            dump_page(me);

        set_term(you);
        c = getch();
        if (c == 'q')      /* wait for user to read it */
            done();
        if (c == ' ')
            dump_page(you);
        sleep(1);
    }
}

dump_page(term)
struct screen *term;
{
    int line;

    set_term(term);
    move(0, 0);
    for (line=0; line<LINES-1; line++) {
        if (fgets(linebuf, sizeof linebuf, fd) == NULL) {
            clrtoebot();

```

```
        done();
    }
    mvprintw(line, 0, " %s" , linebuf);
}
standout();
mvprintw(LINES-1, 0, "--More--");
standend();
refresh();          /* sync screen */
}

/*
 * Clean up and exit.
 */
done()
{
    /* Clean up first terminal */
    set_term(you);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();      /* clear bottom line */
    refresh();        /* flush out everything */
    endwin();         /* curses cleanup */

    /* Clean up second terminal */
    set_term(me);
    move(LINES-1,0); /* to lower left corner */
    clrtoeol();      /* clear bottom line */
    refresh();        /* flush out everything */
    endwin();         /* curses cleanup */

    exit(0);
}
```

EXAMPLE PROGRAM 'window'

The main display of this program is kept in **stdscr**. When the user temporarily wants to put something else on the screen, a new window is created covering part of the screen. A call to **wrefresh** on that window causes the window to be written over **stdscr** on the screen. Calling **refresh** on **stdscr** results in the original window being redrawn on the screen. Note the calls to **touchwin** before writing out an overlapping window. These are necessary to defeat an optimization in **curses**. If you have trouble refreshing a new window which overlaps an old window, it may be necessary to call **touchwin** on the new window to get it completely written out.

```
#include <curses.h>

WINDOW *cmdwin;

main()
{
    int i, c;
    char buf[120];

    initscr();
    nonl();
    noecho();
    cbreak();

    cmdwin = newwin(3, COLS, 0, 0); /* top 3 lines */
    for (i=0; i<LINES; i++)
        mvprintw(i, 0, " This is line %d of stdscr", i);

    for (;;) {
        refresh();
        c = getch();
        switch (c) {
            case 'c': /* Enter command from keyboard */
                werase(cmdwin);
                wprintw(cmdwin, " Enter command:" );
                wmove(cmdwin, 2, 0);
                for (i=0; i<COLS; i++)
                    waddch(cmdwin, '-');
                wmove(cmdwin, 1, 0);
```

```
        touchwin(cmdwin);
        wrefresh(cmdwin);
        wgetstr(cmdwin, buf);
        touchwin(stdscr);
        /*
         * The command is now in buf.
         * It should be processed here.
         */
        break;
    case 'q':
        endwin();
        exit(0);
    }
}
```


Index

A

adding terminal	4-2
	4-3
ASCII characters	6-4
attributes.....	2-11
	3-10
attributes, low level	6-7
attributes, OFF	2-12
	3-11
attributes, ON	2-12
	3-10
audible signal	2-8

B

basic capabilities	7-5
baudrate	5-2
begin pad.....	3-12
begin window.....	3-1
beginning	2-3
bell	2-8
box, window	3-4

C

character representation.....	2-23
character, deleting	2-10
	3-10
character, getting	2-15
	3-11

INDEX

character, inserting	2-9
	2-11
	3-9
clear partial screen	2-6
clear partial window	3-8
clear screen	2-5
	2-20
clear, window	3-8
command line	1-2
comment designation	1-4
compatibility, termcap and terminfo	7-13
compiling terminal descriptions	7-9
control character printing	2-23
conventions	1-3
current screen	2-1
current terminal	4-2
current terminal, change	4-2
	4-4
curscr	2-1
curses description	1-1
curses structure	2-1
cursor optimization	2-7
cursor position	2-4
	2-22
	3-8
cursor position, low level	6-4
cursor position, window	3-12
cur_term	6-6
	6-7

D

defining capabilities	7-3
delay, low level	6-8
delays	2-16
delete line	2-8
	3-9
description characters	7-4
description symbols	7-4
description syntax	7-4

E

echo	2-17
erase	2-5
erase character	5-1
erase, window	3-8
ERR	2-1
exiting	2-3

F

feature description	1-1
flash	2-8
flush	5-2
flush tty driver	2-23
function prefixes	1-4

G

guide organization	1-2
--------------------------	-----

H

highlighting	2-11
	3-10

I

initialization	2-3
initialization, terminal	4-2
	4-3
initialization, terminfo level	6-2
	6-5
input	2-14
input/output functions	2-4
input/output functions, window	3-8
insert line	2-8
	3-9
	I-3

K

keyboard entered capabilities	7-6
keypad	2-22
kill character	5-1

L

longname	2-17
low level usage	6-1
lower level functions	6-1

M

manual pages	1-3
mode setting	2-17
move	2-4
move, in window	3-8
moving windows	3-2
multiple terminals	4-1
multiple terminals, format	4-2
multiple windows	3-5

N

naming terminal	7-2
newline	2-17

O

OK	2-1
option setting	2-20
output	2-6
outputting capabilities, low level	6-8
overlay	3-2
overwrite	3-2

P

pad	3-12
pad initialization	3-12
pad manipulation	3-12
pad, begin	3-12
padding	6-3
	6-4
	6-7
parameter information, low level	6-4
pipe symbol	7-2
portability functions	5-1
preparing terminal descriptions	7-5
print	2-4
print, window	3-8
process ID	4-1
program structure	2-1

R

redraw pad	3-12
redraw screen	2-4
redraw window	3-5
removing windows	3-1
restore tty modes	4-3

S

save tty modes	4-3
scanning	2-15
	3-11
scroll	2-17
setting options	2-20
setting terminal	2-17
special characters	6-4
standard screen	2-1
stdscr	2-1
string, getting	2-15
	3-11
string, inserting	2-9
	3-10

T

TERM.....	4-1
	4-2
	6-5
termcap.....	7-3
	7-13
terminal.....	7-10
terminal descriptions, capabilities	7-3
terminal descriptions, compiling	7-9
terminal descriptions, preparing	7-5
terminal mode setting	2-17
terminal name	2-17
terminal naming	7-2
terminal speed	5-2
terminal, initialization	4-2
	4-3
terminfo database	7-13
terminfo database, description	7-1
terminfo description	1-1
terminfo level	6-1
	6-4
terminfo level, begin	6-2
	6-5
testing an entry	7-10
testing terminal descriptions	7-10
tic — terminfo compiler	7-9
tparm	6-4
	6-7
tput, command	7-11
tputs	6-3
	6-7
tty modes	4-3
	6-6

U

update pad	3-12
update screen	2-4
update window	3-5

V

variables	1-3
video attributes	2-11
	3-10

W

window initialization.....	3-1
window, begin	3-1
window, delete	3-1
window, moving.....	3-2
windows	3-1

